CENGAGE Learning

Gary J. Bronson

C++

PARA INGENIERÍA Y CIENCIAS

Segunda edición

C++ para ingeniería y ciencias

Gary J. Bronson

Editor colaborador:

G.J. Borse, Lehigh University

CENGAGE Learning™

Australia • Brasil • Corea • España • Estados Unidos • Japón • México • Reino Unido • Singapur

CENGAGE Learning™

C++ para ingeniería y ciencias, 2a. ed.
Gary J. Bronson

Presidente de Cengage Learning Iberoamérica:
Javier Arellano Gutiérrez

Director general México y Centroamérica:
Pedro Turbay Garrido

Director editorial y de producción Latinoaméricana:
Raúl D. Zendejas Espejel

Cordinadora editorial:
María Rosas López

Editor de desarrollo:
Pedro de la Garza Rosales

Editora de producción:
Abril Vega Orozco

Diseño de portada:
Perla Alejandra López Romo

© D.R. 2007 por Cengage Learning Editores, S.A. de C.V., una Compañía de Cengage Learning, Inc.
Corporativo Santa Fe
Av. Santa Fe núm. 505, piso 12
Col. Cruz Manca, Santa Fe
C.P. 05349, México, D.F.
Cengage Learning™ es una marca registrada usada bajo permiso.

DERECHOS RESERVADOS. Ninguna parte de este trabajo amparado por la Ley Federal del Derecho de Autor, podrá ser reproducida, transmitida, almacenada o utilizada en cualquier forma o por cualquier medio, ya sea gráfico, electrónico o mecánico, incluyendo, pero sin limitarse a lo siguiente: fotocopiado, reproducción, escaneo, digitalización, grabación en audio, distribución en Internet, distribución en redes de información o almacenamiento y recopilación en sistemas de información a excepción de lo permitido en el Capítulo III, Artículo 27 de la Ley Federal del Derecho de Autor, sin el consentimiento por escrito de la Editorial.

Traducido del libro:
C++ for Engineers and Scientists, 2nd. ed.
publicado en inglés por Course Technology
© 2006
ISBN 0-534-99380-X

Datos para catalogación bibliográfica:
Bronson, Gary J.,
C++ para ingeniería y ciencias, 2a edición.
ISBN-10: 607-481-437-6
ISBN-13: 978-607-481-437-8

Visite nuestro sitio en:
http://latinoamerica.cengage.com

CONTENIDO

CAPÍTULO 1. INTRODUCCIÓN **1**

1.1 INTRODUCCIÓN A LA PROGRAMACIÓN 2
1.2 SOLUCIÓN DE PROBLEMAS Y DESARROLLO DE SOFTWARE 9
1.3 ALGORITMOS 17
1.4 ERRORES COMUNES DE PROGRAMACIÓN 22
1.5 RESUMEN DEL CAPÍTULO 23
1.6 APÉNDICE DEL CAPÍTULO: HARDWARE DE COMPUTACIÓN Y CONCEPTOS DE ALMACENAMIENTO 24

PARTE UNO: PROGRAMACIÓN ORIENTADA A PROCEDIMIENTOS EN C++

CAPÍTULO 2. SOLUCIÓN DE PROBLEMAS MEDIANTE C++ **33**

2.1 INTRODUCCIÓN A C++ 34
2.2 ESTILO DE PROGRAMACIÓN 45
2.3 TIPOS DE DATOS 50
2.4 OPERACIONES ARITMÉTICAS 62
2.5 VARIABLES E INSTRUCCIONES DE DECLARACIÓN 70
2.6 APLICACIÓN DEL PROCEDIMIENTO DE DESARROLLO DE SOFTWARE 85
2.7 APLICACIONES 91
2.8 ERRORES COMUNES DE PROGRAMACIÓN 100
2.9 RESUMEN DEL CAPÍTULO 102

CAPÍTULO 3. ASIGNACIÓN, FORMATEO Y ENTRADA INTERACTIVA **105**

3.1 OPERACIONES DE ASIGNACIÓN 106
3.2 DAR FORMATO A NÚMEROS PARA LA SALIDA DEL PROGRAMA 122
3.3 EMPLEO DE LA BIBLIOTECA DE FUNCIONES MATEMÁTICAS 137
3.4 ENTRADA DE DATOS AL PROGRAMA USANDO EL OBJETO `cin` 147
3.5 CONSTANTES SIMBÓLICAS 161
3.6 APLICACIONES 167

3.7 Errores comunes de programación 177
3.8 Resumen del capítulo 178
3.9 Un acercamiento más a fondo: Errores de programación 180

Capítulo 4. Estructuras de selección 185
4.1 Criterios de selección 186
4.2 La instrucción `if-else` 193
4.3 Instrucciones `if` anidadas 206
4.4 La instrucción `switch` 216
4.5 Aplicaciones 222
4.6 Errores comunes de programación 230
4.7 Resumen del capítulo 231
4.8 Apéndice del capítulo: Un acercamiento más a fondo a la prueba en programación 234

Capítulo 5. Instrucciones de repetición 237
5.1 Estructuras básicas del ciclo 238
5.2 Ciclos `while` 241
5.3 Ciclos `while` interactivos 250
5.4 Ciclos `for` 263
5.5 Técnicas de programación con ciclos 276
5.6 Ciclos anidados 284
5.7 Ciclos `do while` 289
5.8 Errores comunes de programación 293
5.9 Resumen del capítulo 294

Capítulo 6. Modularidad con el uso de funciones 299
6.1 Declaraciones de funciones y parámetros 300
6.2 Devolver un solo valor 321
6.3 Devolver valores múltiples 332
6.4 Apicaciones 342
6.5 Alcance de una variable 361

6.6 Clases de almacenamiento de variables 370
6.7 Errores comunes de programación 379
6.8 Resumen del capítulo 380

Capítulo 7. Completar lo básico 385
7.1 Manejo de excepciones 386
7.2 La clase `string` 392
7.3 Métodos de manipulación de caracteres 410
7.4 Validación de datos de entrada 423
7.5 Espacio de nombres y creación de una biblioteca personal 431
7.6 Errores comunes de programación 437
7.7 Resumen del capítulo 437

Parte dos: Programación orientada a objetos

Capítulo 8. Flujos de archivos de E/S y archivos de datos 443
8.1 Objetos y métodos en el flujo de archivos de E/S 444
8.2 Lectura y escritura de archivos basados en caracteres 459
8.3 Excepciones y comprobación de archivos 471
8.4 Archivos de acceso aleatorio 480
8.5 Flujos de archivo como argumentos de funciones 484
8.6 Errores comunes de programación 488
8.7 Resumen del capítulo 488
8.8 Complemento del capítulo: La biblioteca de clase `iostream` 490

Capítulo 9. Introducción a las clases 495
9.1 Tipos de datos abstractos en C++ (clases) 496
9.2 Constructores 513
9.3 Aplicaciones 522
9.4 Errores comunes de programación 533
9.5 Resumen del capítulo 534

CAPÍTULO 10. FUNCIONES DE CLASES Y CONVERSIONES 537

10.1 ASIGNACIÓN 538
10.2 CARACTERÍSTICAS ADICIONALES DE LA CLASE 547
10.3 FUNCIONES OPERADOR 558
10.4 CONVERSIONES DEL TIPO DE DATOS 567
10.5 APLICACIÓN: SIMULACIÓN DE UNA BOMBA DE GASOLINA CON OBJETOS MÚLTIPLES 576
10.6 HERENCIA DE CLASE 585
10.7 POLIMORFISMO 593
10.8 ERRORES COMUNES DE PROGRAMACIÓN 598
10.9 RESUMEN DEL CAPÍTULO 599

PARTE TRES: ESTRUCTURAS DE DATOS

CAPÍTULO 11. ARREGLOS 605

11.1 ARREGLOS UNIDIMENSIONALES 606
11.2 INICIALIZACIÓN DE ARREGLOS 617
11.3 DECLARACIÓN Y PROCESAMIENTO DE ARREGLOS BIDIMENSIONALES 620
11.4 APLICACIONES 627
11.5 ARREGLOS COMO ARGUMENTOS 635
11.6 LA CLASE DE VECTOR STL 644
11.7 ERRORES COMUNES DE PROGRAMACIÓN 651
11.8 RESUMEN DEL CAPÍTULO 652
11.9 APÉNDICE DEL CAPÍTULO: BÚSQUEDA Y ORDENAMIENTO 653

CAPÍTULO 12. APUNTADORES 667

12.1 DIRECCIONES Y APUNTADORES 668
12.2 NOMBRES DE ARREGLOS COMO APUNTADORES 680
12.3 ARITMÉTICA DE APUNTADORES 689
12.4 TRANSMISIÓN DE DIRECCIONES 693

12.5 Errores comunes de programación 707
12.6 Resumen del capítulo 709

Capítulo 13. Estructuras 711
13.1 Estructuras sencillas 712
13.2 Arreglos de estructuras 718
13.3 Estructuras como argumentos de función 723
13.4 Listas vinculadas 733
13.5 Asignación dinámica de estructuras de datos 741
13.6 Uniones 748
13.7 Errores comunes de programación 751
13.8 Resumen del capítulo 752

Parte cuatro: Temas adicionales

Capítulo 14. Métodos numéricos 755
14.1 ntroducción al cálculo de raíces 756
14.2 El método de bisección 759
14.3 Refinamientos al método de bisección 765
14.4 El método de la secante 775
14.5 Introducción a la integración numérica 778
14.6 La regla trapezoidal 779
14.7 La regla de Simpson 784
14.8 Errores comunes de programación 788
14.9 Resumen del capítulo 788

Capítulo 15. Operaciones bit por bit 791
15.1 El operador AND 792
15.2 El operador OR inclusivo 795

15.3 El operador OR exclusivo 797
15.4 El operador de complemento 799
15.5 Elementos de datos de tamaño diferente 800
15.6 Los operadores de desplazamiento 801
15.7 Resumen del capítulo 806

Apéndice A: Tabla de precedencia de operadores 807

Apéndice B: Códigos ASCII de caracteres 809

Apéndice C: Almacenamiento para un número en punto flotante 811

Apéndice D: Argumentos de línea de comandos 815

Índice 821

PREFACIO

El lenguaje de programación C++, el cual incluye a C como un subconjunto propio, se ha convertido en el lenguaje de programación preeminente en los campos científico y de ingeniería. Sin embargo, para la mayoría de los ingenieros y científicos, emplear el potencial completo de C++, el cual es un lenguaje híbrido que contiene tanto características estructuradas como orientadas a objetos, implica un refinamiento gradual de sus habilidades de programación desde un enfoque estructurado hasta uno orientado a objetos. Una de las razones para ello es que muchos problemas de ingeniería y científicos se pueden resolver de manera eficiente y conveniente empleando únicamente los elementos estructurados de C++.

Este enfoque de refinamiento, de la programación estructural a la orientada a objetos, es el que se adopta en este texto. Por tanto, como la primera edición, esta nueva edición proporciona de entrada un cimiento sólido en programación estructurada. Estos fundamentos se expanden luego a un enfonque orientado a objetos completo dentro de los confines de una progresión pedagógica sólida y accesible. Además, para mantenerla al corriente con el estándar ANSI/ISO para el C++ actual, esta segunda edición presenta diversos cambios significativos y características añadidas. Éstos incluyen lo siguiente:

- Uso de la biblioteca iostream ANSI/ISO para C++ y el mecanismo espacios de nombres en todos los programas
- La presentación del manejo de excepciones en una sección completa, con aplicaciones prácticas de manejo de excepciones presentadas a través del texto
- Presentación de la nueva clase de cadenas C++
- Una exposición a fondo de la validación de los datos de entrada y funciones tanto para verificar el tipo de datos numéricos de un elemento de entrada como para proporcionar la reintroducción de tipos numéricos inválidos
- Una sección nueva por completo dedicada a la clase vectorial Standard Template Library (STL)

Los requerimientos básicos de esta segunda edición, sin embargo, siguen siendo los mismos que en la edición anterior: presentar los temas de una manera clara, sin ambigüedad y accesible para estudiantes principiantes dentro de la estructura de un salón de clases. Con este fin, los elementos centrales de la edición anterior permanecen en esencia sin cambios. Por tanto, la mayor parte de los temas, ejemplos, explicaciones y figuras de la primera edición, excepto por haber sido actualizadas a los estándares ANSI actuales, se encontrarán en esta edición. Los estudiantes deberán estar familiarizados con el álgebra fundamental, pero no se suponen otros prerrequisitos.

En términos prácticos, este texto se ha escrito como apoyo de un curso técnico de programación C++ de uno o dos semestres. Además, está estructurado de manera flexible para que los profesores puedan abordarlo individualmente para la presentación de los temas. Esto se logra de las siguientes maneras.

Excluyendo el capítulo 1, el cual presenta material de nociones de informática para aquellos que requieren estos antecedentes, la parte I presenta los temas de sintaxis estructurada, control de flujo y modularidad básicos necesarios para una comprensión efectiva de las características estructuradas de C++. Cuando se agregan los temas de arreglos (capítulo 11), archivos (capítulo 8) y clases (capítulo 9) a este material, se habrá completado un curso detallado de un semestre. El orden de presentación de estos últimos tres temas queda por entero a la discreción del profesor, ya que cada uno de estos capítulos se ha escrito especialmente para depender sólo del material contenido en la parte I. Además, dependiendo del tiempo y la inclinación, las técnicas numéricas presentadas en el capítulo 14 también pueden presentarse en cualquier punto después de haber completado la parte I. La figura 1 ilustra esta dependencia entre temas para un curso de un semestre.

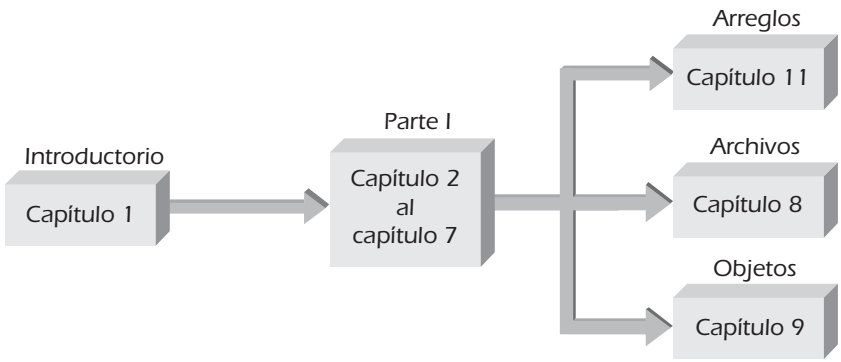


Figura 1. *Dependencia entre temas para un curso de un semestre.*

Una característica específica de este texto es que la parte II, sobre programación orientada a objetos, y la parte III, sobre estructuras de datos, son intercambiables. Por tanto, si desea presentar primero programación orientada a objetos, seguiría una progresión en la secuencia parte I, parte II y parte III. Por otra parte, si desea continuar con un reforzamiento adicional de programación estructurada y presentar la orientación a objetos ya sea al final del curso o al inicio de un segundo semestre, usaría la secuencia parte I, parte III, parte II. En cualquier caso, el material sobre archivos presentado en el capítulo 8, el material sobre arreglos presentado en el capítulo 11, el material sobre clases presentado en el capítulo 9 y el material sobre técnicas numéricas presentado en el capítulo 14 pueden ser introducidos en cualquier momento después de la parte I. La figura 2 presenta una gráfica de la dependencia entre temas para el texto completo, e ilustra la flexibilidad para incluir varios temas dentro de la cobertura general de la programación estructurada, programación orientada a objetos y estructuras de datos.

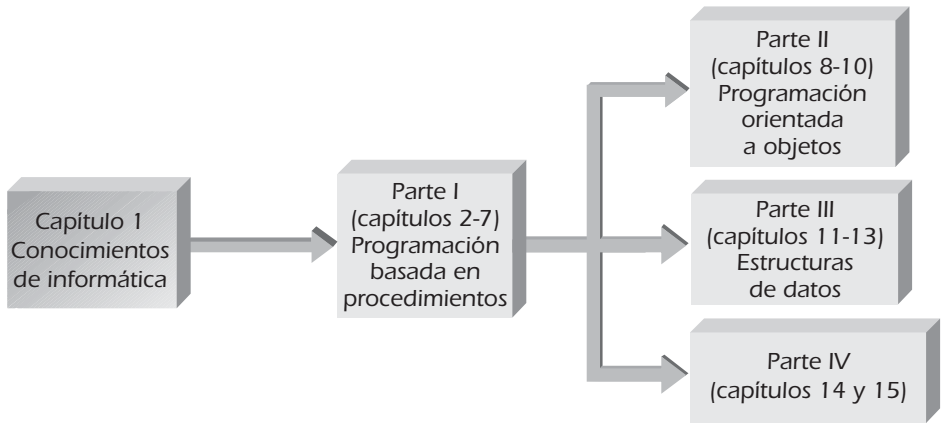


Figura 2. *Gráfica de dependencia entre temas.*

Características distintivas de este libro

Estilo de redacción

Algo muy importante que he encontrado en mis propias clases, es que una vez que el profesor establece el escenario en clase, el libro de texto asignado debe continuar animando, nutriendo y asistiendo al estudiante en la adquisición y "apropiación" del material presentado. Para lograr esto el texto debe estar escrito de una manera que tenga sentido para el estudiante. Mi preocupación principal, y una de las características distintivas de este libro, es que ha sido escrito para el estudiante. Por ello, ante todo, el estilo de redacción usado para transmitir los conceptos presentados es uno de los aspectos más importantes de este texto.

Modularidad

Para producir programas legibles y sostenibles, es esencial la modularidad. C++, por su naturaleza, es un lenguaje modular. Por tanto, inicialmente se realiza la conexión entre las funciones y módulos de C++ en el texto, en la sección 2.1, y se mantiene a través del libro. La idea de la transmisión de parámetros a módulos también se hace de manera temprana, en la sección 3.3, utilizando la biblioteca matemática de C++. Esto introduce a los estudiantes a las funciones y paso de argumentos como técnicas de programación naturales. Con la introducción de técnicas de programación orientadas a objetos en el capítulo 9, el concepto básico de encapsular tanto los datos como las funciones continúa y fortalece este énfasis modular.

Ingeniería de software

En vez de sólo encauzar a los estudiantes a la programación en C++, este texto los introduce a los fundamentos de la ingeniería de software, tanto desde un punto de vista estructurado como orientado a objetos. La sección 2.1 aborda el ciclo de desarrollo de software, presentando uno de los temas principales del texto, el cual es enfatizar las técnicas de solución de problemas. Por tanto, se establece desde el principio la importancia de entender y definir un problema, seleccionar y refinar una solución, y comprender la relación entre análisis, diseño, codificación y prueba, y se continúa con ejemplos prácticos en todas las secciones de aplicaciones subsiguientes.

Aplicaciones

Empezando con el capítulo 2, cada capítulo contiene una sección de aplicaciones con un promedio de dos aplicaciones completas por capítulo. Cada aplicación demuestra y refuerza la solución de problemas efectiva dentro del contexto del ciclo de desarrollo de software introducido en la sección 1.2, y se extiende al desarrollo orientado a objetos cuando se introducen las clases en el capítulo 9.

Prueba de programas

Cada programa C++ en este texto ha sido compilado, ejecutado y probado con éxito para asegurar la calidad usando Microsoft Visual C++.NET. El código fuente para todos los programas puede encontrarse en el sitio Web *http://www.course.com*. Esto permitirá a los estudiantes experimentar y expandir los programas existentes y modificarlos con facilidad como lo requieren diversos ejercicios al final de las secciones.

Características pedagógicas

Para lograr que C++ sea accesible como un curso de primer nivel, se han incorporado al texto las siguientes características pedagógicas:

Ejercicios de final de sección Casi todas las secciones del libro contienen numerosos y diversos ejercicios para el desarrollo de habilidades y programación. Además, se proporcionan las soluciones a todos los ejercicios en la sección Instructor Downloads en *http://www.course.com*.

Errores comunes de programación y resumen del capítulo

Cada capítulo termina con una sección sobre errores comunes de programación y un resumen de los temas principales cubiertos en el capítulo.

Secciones de enriquecimiento

Dados los muchos énfasis diferentes que pueden aplicarse en la enseñanza de C++, se han incluido diversas secciones de enriquecimiento como apéndices de los capítulos. Esto permite proveer distintos énfasis con diferentes estudiantes de diversas clases de C++.

Notas de programación

Las notas de programación son un conjunto separado de recuadros sombreados que resaltan técnicas de programación alternativas y avanzadas, puntos técnicos útiles, consejos de programación y trucos de programación utilizados por programadores profesionales.

Recuadros de punto de información

Éstos son un conjunto de recuadros sombreados que presentan aclaraciones breves adicionales de conceptos comunes o difíciles, como abstracción, lvalues y rvalues, valores frente a identidades, marcadores y formateo de líneas.

Descripciones de seudocódigo

Se utiliza seudocódigo a través de todo el texto. Se presentan símbolos de diagramas de flujo pero sólo se usan para presentar en forma visual los constructos de control de flujo.

Opciones de carrera

Se proporciona un conjunto de recuadros sombreados que describen varias opciones para algunas carreras de ingeniería, como eléctrica, química, mecánica y aeronáutica.

Apéndices y complementos

Se proporciona un conjunto amplio de apéndices. Éstos incluyen apéndices sobre Precedencia de Operadores, Códigos ASCII y Operaciones de Bits. Adicionalmente se proporcionan varios manuales para usar los diversos compiladores C++, los cuales pueden localizarse en *http://www.course.com*, junto con archivos electrónicos con las soluciones a todos los ejercicios de programación y archivos con los códigos fuente.

A Rochelle, Matthew, Jeremy y David

RECONOCIMIENTOS

La redacción de esta segunda edición es el resultado directo del éxito (y limitaciones) de la primera edición. A este respecto, mi reconocimiento y aprecio más sinceros es para los instructores y estudiantes que encontraron que aquella edición les fue útil en sus respectivas búsquedas para enseñar y aprender C++.

A continuación, me gustaría agradecer a Alyssa Pratt, mi gerente de producto en Course Technology. Además de su fe y aliento continuos, sus ideas y asociación fueron decisivas en la creación de este texto. Una vez que se completó el proceso de redacción, la tarea de convertir el manuscrito final en un libro de texto dependió de muchas personas aparte de mí. Por ello deseo agradecer en especial a mi editor en GEX Publishing Services, el diseñador de interiores, y Jennifer Roehrig de GEX Publishing Services. La dedicación de este equipo de personas fue importante en extremo para mí y estoy muy agradecido con ellos. Como siempre, cualquier error en el texto descansa solamente en mis hombros.

Como con la primera edición, hago llegar un reconocimiento especial al doctor G.J. Borse de Lehigh University, quien proporcionó material que fue adaptado para este texto. De manera específica, su contribución incluye casi todo el capítulo 14 y la información contenida dentro de los recuadros Opciones de Carrera, los cuales el doctor Borse me permitió amablemente adaptar de su texto FORTRAN 77 (derechos reservados por PWS Publishing). Además, deseo agradecer a Karen Langbert por proporcionar conocimientos específicos y verificación matemática en diversas aplicaciones de ingeniería.

También me gustaría reconocer, con gratitud extrema, el aliento y apoyo directos de Fairleigh Dickinson University. De manera específica, esto incluye el clima académico positivo proporcionado por el rector, doctor Kenneth Greene, y el aliento y apoyo directos para hacer uso de este ambiente de mi decano, doctor David Steele, y mi presidente, doctor Paul Yoon. Sin su apoyo, este texto no podría haberse escrito.

Por último, aprecio en forma profunda la paciencia, comprensión y amor proporcionados por mi amiga, esposa y socia, Rochelle.

Gary Bronson
2005

CAPÍTULO 1
Introducción

TEMAS

1.1 INTRODUCCIÓN A LA PROGRAMACIÓN
- LENGUAJE DE MÁQUINA
- LENGUAJES ENSAMBLADORES
- LENGUAJES DE NIVELES BAJO Y ALTO
- ORIENTACIONES A PROCEDIMIENTOS Y A OBJETOS
- SOFTWARE DE APLICACIÓN Y DE SISTEMA
- EL DESARROLLO DE C++

1.2 SOLUCIÓN DE PROBLEMAS Y DESARROLLO DE SOFTWARE
- FASE I. DESARROLLO Y DISEÑO
- FASE II. DOCUMENTACIÓN
- FASE III. MANTENIMIENTO
- RESPALDO

1.3 ALGORITMOS

1.4 ERRORES COMUNES DE PROGRAMACIÓN

1.5 RESUMEN DEL CAPÍTULO

1.6 APÉNDICE DEL CAPÍTULO: HARDWARE DE COMPUTACIÓN Y CONCEPTOS DE ALMACENAMIENTO
- ALMACENAMIENTO DE COMPUTADORA
- NÚMEROS EN COMPLEMENTO A DOS
- PALABRAS Y DIRECCIONES
- CONSIDERACIÓN DE LAS OPCIONES DE CARRERA: INGENIERÍA AERONÁUTICA Y AEROESPACIAL

1.1 INTRODUCCIÓN A LA PROGRAMACIÓN

Una computadora es una máquina y, como otras máquinas, como un automóvil o una podadora, debe encenderse y luego conducirse, o controlarse, para hacer la tarea que se pretende realizar. En un automóvil, por ejemplo, el control es proporcionado por el conductor, quien se sienta en su interior y lo dirige. En una computadora, el conductor es un conjunto de instrucciones llamado programa. De manera más formal, un **programa de computadora** es un conjunto independiente de instrucciones usado para operar una computadora con el fin de producir un resultado específico. Otro término para un programa o conjunto de programas es **software**, y se usarán ambos términos de manera indistinta a través del texto.[1]

El proceso de escribir un programa, o software, se llama **programación**, mientras al conjunto que puede usarse para construir un programa se llama **lenguaje de programación**. Los lenguajes de programación disponibles se presentan en una variedad de formas y tipos.

Lenguaje de máquina

En su nivel más fundamental, los únicos programas que pueden usarse en realidad para operar una computadora son los **programas en lenguaje de máquina**. Tales programas, los cuales también se conocen como **programas ejecutables**, o **ejecutables** para abreviar, consisten en una secuencia de instrucciones compuestas por números binarios como:[2]

```
11000000 000000000001 000000000010
11110000 000000000010 000000000011
```

Estas instrucciones en lenguaje de máquina constan de dos partes: una de instrucción y una de dirección. La parte de instrucción, a la cual se conoce como **opcode** (abreviatura de "código de operación"), por lo general es el conjunto de bits en el extremo izquierdo de la instrucción y le indica a la computadora la operación a realizar, como sumar, restar, multiplicar, etc., mientras los bits en el extremo derecho especifican las direcciones de memoria de los datos que se van a usar. Por ejemplo, suponiendo que los ocho bits en el extremo izquierdo de la primera instrucción enlistada antes contienen el código de operación para sumar, y los siguientes dos grupos de doce bits son las direcciones de los dos operandos que se van a sumar, esta instrucción sería un comando para "sumar los datos en la ubicación 1 de la memoria a los datos en la ubicación 2 de la memoria". Del mismo modo, suponiendo que el opcode 11110000 significa multiplicar, la siguiente instrucción es un comando para "multiplicar los datos en la ubicación 2 de la memoria por los datos en la ubicación 3". (La sección 1.6 explica cómo convertir de números binarios a decimales.)

Lenguajes ensambladores

Debido a que cada clase de computadora, como IBM, Apple y Hewlett Packard, tiene su propio lenguaje de máquina particular, es muy tedioso y tardado escribir esos programas en lenguaje de máquina.[3] Uno de los primeros avances en la programación fue la sustitución de

[1] De una manera más incluyente, el término software también se usa para denotar tanto los programas como los datos con los que operarán los programas.

[2] Revise la sección 1.6 al final de este capítulo si no está familiarizado con los números binarios.

[3] En la actualidad, el lenguaje en el nivel de máquina está definido por el procesador alrededor del cual está construida la computadora.

símbolos en forma de palabras, como ADD, SUB, MUL, por los opcodes binarios y los números decimales y etiquetas por las direcciones en memoria. Por ejemplo, usando estos símbolos y valores decimales para las direcciones en memoria, las dos instrucciones en lenguaje de máquina anteriores pueden escribirse como:

```
ADD 1, 2
MUL 2, 3
```

Los lenguajes de programación que usan este tipo de notación simbólica se conocen como **lenguajes ensambladores**. Debido a que las computadoras sólo pueden ejecutar programas en lenguaje de máquina, el conjunto de instrucciones contenido dentro de un programa en lenguaje ensamblador debe traducirse a un programa de lenguaje de máquina antes que pueda ejecutarse en una computadora. Los programas traductores que realizan esta función para los programas en lenguaje ensamblador se conocen como **ensambladores** (véase la figura 1.1).

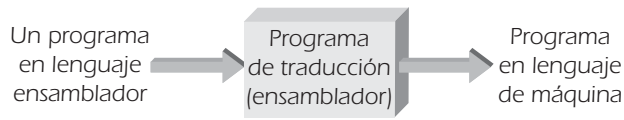


Figura 1.1 *Los programas ensambladores deben traducirse.*

Lenguajes de niveles bajo y alto

Tanto los lenguajes de máquina como los ensambladores se clasifican como **lenguajes de nivel bajo**. Esto se debe a que ambos tipos de lenguaje usan instrucciones que se vinculan en forma directa con un tipo de computadora. Como tal, un programa en lenguaje ensamblador está limitado porque sólo puede usarse con el tipo de computadora específica para el cual se escribió. Sin embargo, estos programas permiten usar las características especiales de un tipo de computadora particular y por lo general se ejecutan en el nivel más rápido posible.

En contraste con los lenguajes de nivel bajo están los lenguajes de alto nivel. Un **lenguaje de alto nivel** usa instrucciones que se parecen a los lenguajes escritos, como el inglés, y pueden ejecutarse en una variedad de tipos de computadora. Visual Basic, C, C++ y Java son ejemplos de lenguajes de alto nivel. Usando C++, una instrucción para sumar dos números y multiplicarlos por un tercer número puede escribirse como:

```
resultado = (primero + segundo) * tercero;
```

Los programas escritos en un lenguaje de computadora (de alto o bajo nivel) se conocen como **programas fuente** y **código fuente**. Una vez que se ha escrito un programa en un lenguaje de alto nivel también debe traducirse, como un programa ensamblador de bajo nivel, al lenguaje de máquina de la computadora en que se va a ejecutar. Esta traducción puede lograrse en dos formas.

Cuando cada declaración en un programa fuente de alto nivel es traducida de manera individual y ejecutada inmediatamente después de la traducción, el lenguaje de programación usado se llama **lenguaje interpretado** y el programa que hace la traducción se llama **intérprete**.

Cuando todas las instrucciones en un programa fuente de alto nivel son traducidas como una unidad completa antes que cualquier declaración sea ejecutada, el lenguaje de programa-

ción usado se llama **lenguaje compilado**. En este caso, el programa que hace la traducción se llama **compilador**. Pueden existir tanto versiones compiladas como interpretadas de un lenguaje, aunque de manera típica predomina una. C++ es predominantemente un lenguaje compilado.

La figura 1.2 ilustra la relación entre un código fuente de C++ y su compilación en un programa ejecutable en lenguaje de máquina. Como se muestra, el programa fuente se introduce usando un programa editor. Éste es en efecto un programa procesador de palabras que es parte del ambiente de desarrollo proporcionado por el compilador. Debe entenderse, sin embargo, que la introducción del código sólo puede comenzar después que una aplicación se ha analizado y comprendido en forma minuciosa y el diseño del programa ha sido planeado con cuidado. La forma en que se logra esto se explica en la siguiente sección.

La traducción del programa fuente C++ en un programa en lenguaje de máquina comienza con el compilador. La salida producida por el compilador se llama **programa objeto**, el cual es una versión en lenguaje de máquina del código fuente. En casi todos los casos, su código fuente usará código preprogramado existente, con código que ha escrito con anterioridad o código proporcionado por el compilador. Éste podría incluir código matemático para encontrar una raíz cuadrada, por ejemplo, o código que se está reutilizando de otra aplicación. Además, un programa C++ grande puede almacenarse en dos o más archivos de programa separados. En todos estos casos, este código adicional debe combinarse con el programa objeto antes que el programa pueda ejecutarse. Es tarea del **ligador** lograr este paso. El resultado del proceso de ligamiento es un programa en lenguaje de máquina completado, que contiene todo el código requerido por el programa, el cual está listo para su ejecución. El último paso en el proceso es cargar este programa en lenguaje de máquina en la memoria principal de su computadora para su ejecución real.

Orientaciones a procedimientos y a objetos

Además de clasificar los lenguajes de programación como de alto o bajo nivel, también se clasifican por su orientación a procedimientos u objetos. En un **lenguaje orientado a procedimientos** las instrucciones disponibles se usan para crear unidades independientes, conocidas como **procedimientos**. El propósito de un procedimiento es aceptar datos como entrada y transformarlos de alguna manera para producir un resultado específico como una salida. Hasta la década de los años 90 la mayor parte de los lenguajes de programación de alto nivel eran orientados a procedimientos.

En la actualidad, un segundo enfoque, la orientación a objetos, ha tomado el escenario central. Una de las motivaciones para **lenguajes orientados a objetos** fue el desarrollo de pantallas gráficas y soporte para las interfaces gráficas de usuario (GUI), capaces de desplegar múltiples ventanas que contienen tanto formas gráficas como texto. En tal ambiente, cada ventana en la pantalla puede considerarse un objeto con características asociadas, como color, posición y tamaño. Usando un enfoque orientado a objetos, un programa debe definir primero los objetos que manipulará, incluyendo una descripción de las características generales de los objetos y unidades específicas para manipularlos, como cambiar el tamaño y la posición y transferir datos entre objetos. Es de igual importancia que los lenguajes orientados a objetos tiendan a soportar la reutilización del código existente con más facilidad, lo cual elimina la necesidad de revalidar y reexaminar código nuevo o modificado. C++, el cual se clasifica como un lenguaje orientado a objetos, contiene características que se encuentran en los lenguajes orientados a procedimientos y a objetos. En este texto se diseñarán, desarrollarán y presentarán ambos tipos de código, que es la forma en que se escribe la mayor parte de los programas C++ actuales. Debido a que el código C++ orientado a objetos siempre contiene algún código

de procedimientos, y muchos programas C++ simples se escriben por completo usando sólo código de procedimientos, este tipo de código se presenta primero.

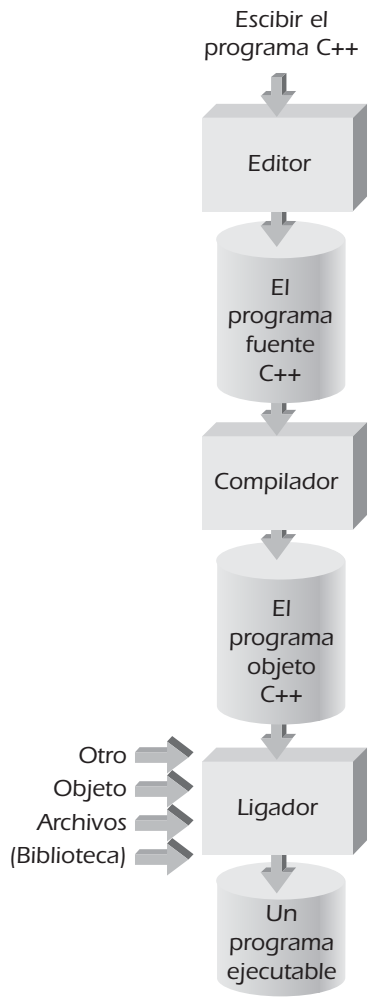


Figura 1.2 Creación de un programa C++ ejecutable.

Software de aplicación y de sistema

El software de aplicación y el software de sistema son dos categorías lógicas de programas de computadora. El **software de aplicación** consiste en aquellos programas escritos para realizar tareas particulares requeridas por los usuarios. Todos los programas en este libro son ejemplos de software de aplicación.

El **software de sistema** es la colección de programas que deben estar disponibles en cualquier sistema de cómputo en el que ha de operar. En los primeros entornos de cómputo de las décadas de los años 50 y 60, el usuario tenía que cargar al inicio el software de sistema en forma manual para preparar la computadora para que hiciera algo. Esto se llevaba a cabo usando hileras de conmutadores en un panel frontal. Se decía que aquellos comandos iniciales

introducidos en forma manual iniciaban (**boot**) la computadora, una expresión derivada de la expresión inglesa *pulling oneself up by the bootstraps* que significa "salir adelante sin ayuda". En la actualidad, el llamado cargador inicial (**bootstrap loader**) es un componente permanente que se ejecuta de manera automática desde el software del sistema de la computadora.

De manera colectiva, el conjunto de programas de sistema usados para operar y controlar una computadora se llama **sistema operativo**. Los sistemas operativos modernos incluyen las siguientes tareas: administración de memoria; asignación de tiempo de CPU; control de unidades de entrada y salida como teclado, pantalla e impresoras, y la administración de todos los dispositivos de almacenamiento secundarios. Muchos sistemas operativos manejan programas grandes y múltiples usuarios, en forma concurrente, dividiendo los programas en segmentos que son movidos entre el disco y la memoria conforme se necesita. Tales sistemas operativos permiten que más de un usuario ejecute un programa en la computadora, lo cual le da a cada usuario la impresión que la computadora y los periféricos son sólo suyos. Esto se conoce como un sistema **multiusuario**. Además, muchos sistemas operativos, incluyendo la mayor parte de los ambientes con ventanas, permiten a cada usuario ejecutar múltiples programas. Dichos sistemas operativos se conocen como sistemas **multiprogramados** y **multitareas**.

El desarrollo de C++

En un nivel básico, el propósito de casi todos los programas de aplicación es procesar datos para producir uno o más resultados específicos. En un lenguaje de procedimientos, un programa se construye a partir de conjuntos de instrucciones, con cada conjunto nombrado como un procedimiento, como se señaló con anterioridad. En efecto, cada procedimiento mueve los datos un paso más cerca de la salida final deseada a lo largo de la ruta mostrada en la figura 1.3.

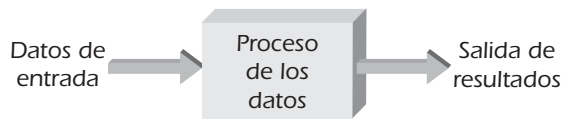


Figura 1.3 Operaciones de procedimiento básicas.

El proceso de programación ilustrado en la figura 1.3 refleja en forma directa las unidades de hardware de entrada, procesamiento y salida usadas para construir una computadora (véase la sección 1.6). Esto no fue accidental porque los primeros lenguajes de programación fueron diseñados de manera específica para corresponder y controlar en forma directa, lo más óptimamente posible, a las unidades de hardware apropiadas.

El primer lenguaje de procedimientos, llamado FORTRAN, cuyo nombre se deriva de *FOR*mula *TRAN*slation, fue introducido en 1957 y siguió siendo popular durante la década de los años 60 y principios de la década de los años 70. (Otro lenguaje de programación de nivel alto desarrollado en forma casi concurrente con FORTRAN, pero que nunca logró la aceptación abrumadora de FORTRAN, fue nombrado ALGOL.) FORTRAN tiene instrucciones tipo álgebra que se concentran en la fase de procesamiento mostrada en la figura 1.3 y fue desarrollado para aplicaciones científicas y de ingeniería que requerían salidas numéricas de gran precisión, incluyendo muchos lugares decimales. Por ejemplo, calcular la trayectoria de un cohete o el nivel de concentración bacteriana en un estanque contaminado, como se ilustra en la figura 1.4, requiere evaluar una ecuación matemática a un alto grado de precisión numérica y es típico de las aplicaciones basadas en FORTRAN.

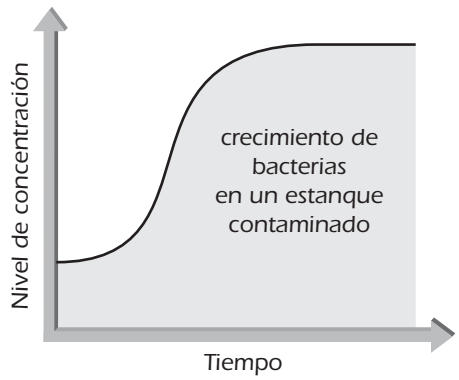


Figure 1.4 *FORTRAN fue desarrollado para aplicaciones científicas y de ingeniería.*

En orden de aparición, el siguiente lenguaje de aplicación de nivel alto significativo fue COBOL, el cual fue introducido en la década de los años 60 y permaneció como un lenguaje de procedimientos importante hasta la década de los años 80. La palabra COBOL se formó por las siglas de COmmon Business-Oriented Language. Este lenguaje tenía características enfocadas hacia aplicaciones de negocios que requerían cálculos matemáticos más simples que los necesarios para aplicaciones de ingeniería. Uno de los beneficios más notables de COBOL fue que proporcionaba formatos de salida extensos que facilitaban la creación de informes que contenían muchas columnas de números y totales en dólares y centavos formateados con esmero, como se ilustra en la figura 1.5. Esto obligó a los programadores a construir procedimientos estructurados bien definidos que seguían un patrón más consistente que el requerido por FORTRAN.

No. de parte	Descripción	Cantidad	Precio
12225	#4 Clavos, normales	25 cajas	1.09
12226	#6 Clavos, normales	30 cajas	1.09
12227	#8 Clavos, normales	65 cajas	1.29
12228	#10 Clavos, normales	57 cajas	1.35
12229	#12 Clavos, normales	42 cajas	
12230	#16 Clavos, normales		

Figure 1.5 *COBOL fue desarrollado para aplicaciones de negocios.*

Otro lenguaje, BASIC (o Beginners All-purpose Symbolic Instruction Code), fue desarrollado en Dartmouth College más o menos al mismo tiempo que COBOL. BASIC era en esencia una versión ligeramente reducida de FORTRAN y pretendía ser un lenguaje introductorio para estudiantes universitarios. Era un lenguaje relativamente sencillo, fácil de entender, que no requería un conocimiento detallado de una aplicación específica. Su principal desventaja era que no requería ni imponía un enfoque consistente o estructurado para crear programas. Con frecuencia, el programador no podía comprender con facilidad qué hacía su programa BASIC después de un tiempo breve.

Para remediar esto y adecuar la programación a una base más científica y racional que hiciera más fácil entender y reutilizar el código, se desarrolló el lenguaje Pascal. (Pascal no es una sigla, sino que se le puso este nombre en honor al matemático del siglo XVII Blaise Pascal.) Introducido en 1971, proporcionó a los estudiantes un fundamento más firme en el diseño de programación estructurada que lo aportado por versiones anteriores de BASIC.

Los programas estructurados se crean usando un conjunto de estructuras bien definidas organizadas en secciones de programación individuales, cada una de las cuales ejecuta una tarea específica que puede probarse y modificarse sin perturbar otras secciones del programa. Sin embargo, el lenguaje Pascal estaba estructurado en forma tan rígida que no existían escapes de las secciones estructuradas cuando hubieran sido útiles. Esto era una limitante para muchos proyectos del mundo real y es una de las razones por las que Pascal no fue aceptado en forma amplia en los campos científico y de ingeniería. En cambio, el lenguaje C, el cual es un lenguaje de procedimientos estructurado desarrollado en la década de los años 70 en AT&T Bell Laboratories por Ken Thompson, Dennis Ritchie y Brian Kernighan, se convirtió en el lenguaje para aplicaciones de ingeniería dominante de la década de los años 80. Este lenguaje tiene un amplio conjunto de capacidades que permite que se escriba como un lenguaje de nivel alto mientras conserva la capacidad de acceso directo a las características del nivel de máquina de una computadora.

C++ fue desarrollado a principios de la década de los años 80, cuando Bjarne Stroustrup (también en AT&T) usó sus conocimientos en lenguaje de simulación para crear un lenguaje de programación orientado a objetos. Una característica central de los lenguajes de simulación es que modelan situaciones de la vida real como objetos. Esta orientación a objetos, la cual era ideal para objetos gráficos presentados en pantalla como rectángulos y círculos, se combinó con características de C, existentes para formar el lenguaje C++. Por tanto, C++ conservó el conjunto extenso de capacidades estructuradas y de procedimientos proporcionadas por C, pero agregó su propia orientación a objetos para convertirse en un verdadero lenguaje de programación de uso general. Como tal, C++ puede usarse desde programas interactivos simples, hasta programas de ingeniería y científicos sofisticados y complejos, dentro del contexto de una estructura en verdad orientada a objetos.

Ejercicios 1.1

1. Defina los siguientes términos:
   - **a.** programa de computadora
   - **b.** programación
   - **c.** lenguaje de programación
   - **d.** lenguaje de alto nivel
   - **e.** lenguaje de bajo nivel
   - **f.** lenguaje de máquina
   - **g.** lenguaje ensamblador
   - **h.** lenguaje orientado a procedimientos
   - **i.** lenguaje orientado a objetos
   - **j.** programa fuente
   - **k.** compilador
   - **l.** intérprete
2. Describa el propósito y usos principales del software de aplicación y de sistema.

3. **a.** Describa la diferencia entre lenguajes de alto y bajo nivel.
b. Describa la diferencia entre lenguajes orientados a procedimientos y a objetos.

4. Describa las semejanzas y diferencias entre ensambladores, intérpretes y compiladores.

5. **a.** Dados los siguientes códigos de operación,

```
11000000 significa sumar el 1er. operando al 2o. operando
10100000 significa restar el 1er. operando del 2o. operando
11110000 significa multiplicar el 2o. operando por el 1er. operando
11010000 significa dividir el 2o. operando entre el 1er. operando
```

traduzca las siguientes instrucciones al español:

opcode	Dirección del 1er. operando	Dirección del 2o. operando
11000000	000000000001	0000000000010
11110000	000000000010	0000000000011
10100000	000000000100	0000000000011
11010000	000000000101	0000000000011

b. Suponiendo que las siguientes ubicaciones contienen los datos proporcionados, determine el resultado producido por las instrucciones listadas en el ejercicio 5a. Para este ejercicio, suponga que cada instrucción es ejecutada de manera independiente de cualquier instrucción.

Dirección	Valor inicial (en decimales) almacenado en esta dirección
00000000001	5
00000000010	3
00000000011	6
00000000100	14
00000000101	4

6. Reescriba las instrucciones en el nivel de máquina enlistadas en el ejercicio 5a usando notación de lenguaje ensamblador. Use los nombres simbólicos ADD, SUB, MUL y DIV para operaciones de adición, sustracción, multiplicación y división, respectivamente. Al escribir las instrucciones use valores decimales para las direcciones.

1.2 Solución de problemas y desarrollo de software

Sin importar cuál campo de trabajo elija o cuál pueda ser su estilo de vida, tendrá que resolver problemas. Muchos de éstos, como sumar el cambio en su bolsillo, pueden resolverse rápido y fácil. Otros, como montar en bicicleta, requieren algo de práctica pero pronto se vuelven automáticos. Otros más requieren de una planeación y premeditación considerables para que la solución sea apropiada y eficiente. Por ejemplo, construir una red telefónica celular o crear un sistema de administración de inventarios para un gran almacén son problemas para los cuales las soluciones por ensayo y error podrían resultar costosas y desastrosas.

Crear un programa no es diferente porque un programa es una solución desarrollada para resolver un problema particular. Por ello, escribir un programa casi es el último paso en un proceso de determinar primero cuál es el problema y el método que se usará para resolverlo. Cada campo de estudio tiene su propio nombre para el método sistemático usado para resolver problemas mediante el diseño de soluciones adecuadas. En las ciencias y la ingeniería el enfoque se conoce como el **método científico**, mientras en el análisis cuantitativo el enfoque se denomina **enfoque de sistemas**.

El método usado por los profesionales que desarrollan software para entender el problema que se va a solucionar y para crear una solución de software efectiva y apropiada se llama **procedimiento de desarrollo de software**. Este procedimiento, como se ilustra en la figura 1.6, consiste en tres fases que se superponen:

- Diseño y desarrollo
- Documentación
- Mantenimiento

Como disciplina, la **ingeniería de software** se encarga de crear programas y sistemas legibles, eficientes, confiables y mantenibles, utilizando el procedimiento de desarrollo de software para lograr esta meta.

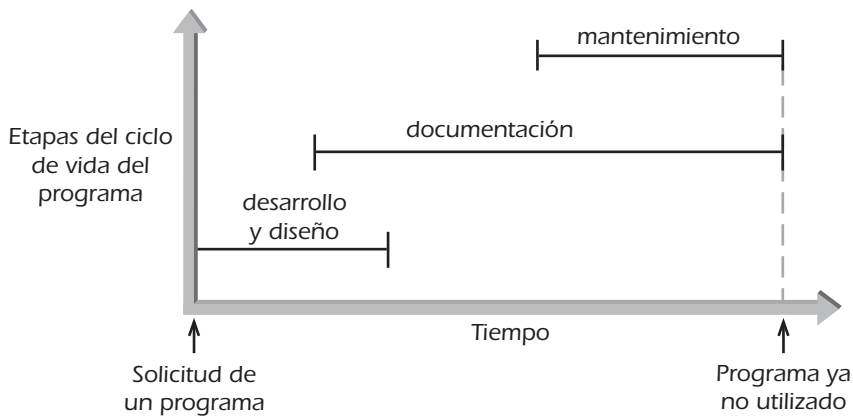


Figura 1.6 *Las tres fases del desarrollo de programas.*

Fase I. Desarrollo y diseño

La fase I comienza con el planteamiento de un problema o con una solicitud específica de un programa, lo cual se conoce como **requerimiento de programa**. Una vez que se ha planteado un problema o se ha hecho una solicitud específica para un programa, comienza la fase de diseño y desarrollo. Esta fase consta de cuatro pasos bien definidos, como se ilustra en la figura 1.7 y se resume a continuación.

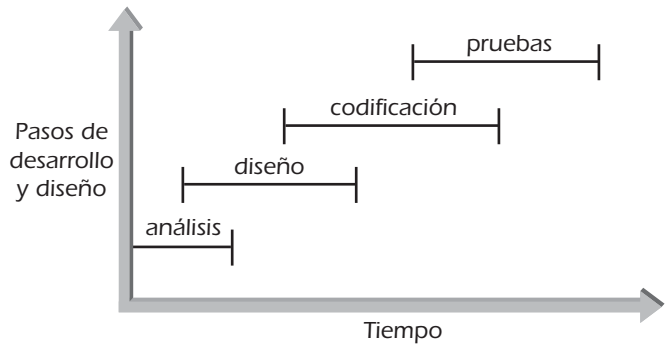


Figura 1.7 Los pasos de diseño y desarrollo.

Paso 1 Analizar el problema

Este paso es necesario para asegurar que el problema está definido y se entiende con claridad. La determinación de que el problema está definido en forma clara se hace sólo después que quien realiza el análisis entiende qué salidas se requieren y qué entradas se necesitarán. Para lograr esto el analista debe tener una comprensión de la forma en que se pueden usar las entradas para producir la salida deseada. Por ejemplo, suponga que recibe la siguiente tarea:

> ***Escriba un programa que nos proporcione la información que necesitamos sobre los círculos. Termínelo para mañana.***
>
> ***— La gerencia***

Un análisis simple de esta solicitud de programa revela que no es un problema bien definido en absoluto, porque no sabemos con exactitud qué información de salida se requiere. Por ello, sería un error enorme comenzar de inmediato a escribir un programa para solucionarlo. Para aclarar y definir el planteamiento del problema, su primer paso deberá ser ponerse en contacto con "La gerencia" para definir con exactitud qué va a producir el programa (sus salidas). Suponga que hizo esto y se enteró que lo que en realidad se deseaba es un programa para calcular y mostrar la circunferencia de un círculo cuando se da el radio. Debido a que existe una fórmula para convertir la entrada en la salida, puede proceder al siguiente paso. Si no se está seguro de cómo obtener la salida requerida o exactamente cuáles entradas se necesitan, se requiere un análisis más profundo. Esto de manera típica significa obtener más información antecedente acerca del problema o aplicación. Con frecuencia también implica hacer uno o más cálculos manuales para asegurar que se entiende qué entradas son necesarias y cómo deben combinarse para lograr la salida deseada.

Innumerables horas se han dedicado a escribir programas de computadora que nunca se han usado o han causado una animosidad considerable entre el programador y el usuario debido a que el programador no produjo lo que el usuario necesitaba o esperaba. Los programadores exitosos entienden y evitan esto al asegurarse que entienden los requerimientos del problema. Éste es el primer paso en la creación de un programa y el más importante,

porque en él se determinan las especificaciones para la solución final del programa. Si los requerimientos no son entendidos por completo antes que comience la programación, los resultados casi siempre son desastrosos.

Por ejemplo, imagine diseñar y construir una casa sin entender por completo las especificaciones del propietario. Después que se ha terminado, el propietario le dice que se requería un baño en el primer piso, donde usted ha construido una pared entre la cocina y el comedor. Además, esa pared en particular es una de las paredes de soporte principales para la casa y contiene numerosas tuberías y cables eléctricos. En este caso, agregar un baño requiere una modificación bastante importante a la estructura básica de la casa.

Los programadores experimentados entienden la importancia de analizar y comprender los requerimientos del programa antes de codificarlo, en especial si también han elaborado programas que después han tenido que desmantelarse y rehacerse por completo. La clave del éxito aquí, la cual a fin de cuentas determina el éxito del programa final, es determinar el propósito principal del sistema visto por la persona que hace la solicitud. Para sistemas grandes, el análisis por lo general es realizado por un analista de sistemas. Para sistemas más pequeños o programas individuales, el análisis de manera típica se lleva a cabo en forma directa por el programador.

Sin tener en cuenta cómo se hizo el análisis, o por quién, al concluirlo deberá haber una comprensión clara de:

- Qué debe hacer el sistema o programa
- Qué salidas debe producir
- Qué entradas se requieren para crear las salidas deseadas

Paso 2 Desarrollar una solución

En este paso, se selecciona el conjunto exacto de pasos, llamado algoritmo, que se usará para resolver el problema. La solución se obtiene de manera típica por una serie de refinamientos, comenzando con el algoritmo inicial encontrado en el paso de análisis, hasta que se obtenga un algoritmo aceptable y completo. Este algoritmo debe verificarse, si no se ha hecho en el paso de análisis, para asegurar que produce en forma correcta las salidas deseadas. Por lo general la verificación se hace realizando uno o más cálculos manuales que no se han hecho.

Para programas pequeños el algoritmo seleccionado puede ser en extremo simple y consistir de sólo uno o más cálculos. De manera más general, la solución inicial debe refinarse y organizarse en subsistemas más pequeños, con especificaciones sobre la forma en que los subsistemas interactuarán entre sí. Para lograr este objetivo, la descripción del algoritmo comienza desde el requerimiento de nivel más alto (superior) y procede en forma descendente a las partes que deben elaborarse para lograr este requerimiento. Para hacer esto más significativo, considere un programa de computadora para dar seguimiento al número de partes en un inventario. La salida requerida para este programa es una descripción de todas las partes que se llevan en el inventario y el número de unidades de cada artículo en existencia; las entradas dadas son la cantidad inicial en inventario de cada parte, el número de artículos vendidos, el número de artículos devueltos y el número de artículos comprados.

Para estas especificaciones, el diseñador podría organizar al principio los requerimientos para el programa en las tres secciones ilustradas en la figura 1.8. Esto se llama ***diagrama de estructura de primer nivel*** porque representa la primera estructura general del programa seleccionado por el diseñador.

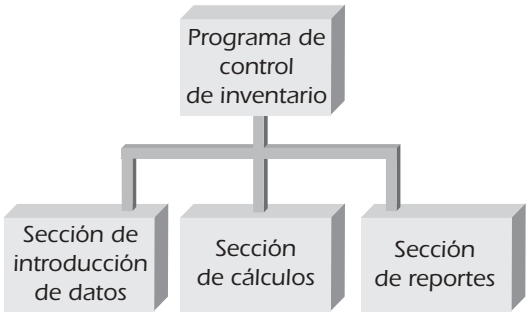


Figura 1.8 *Diagrama de estructura de primer nivel.*

Una vez que se ha desarrollado una estructura inicial, se refina hasta que las tareas indicadas en los cuadros están definidas por completo. Por ejemplo, tanto los módulos de introducción de datos como de reportes que se muestran en la figura 1.8 deberían refinarse más. El módulo de introducción de datos por supuesto debe incluir provisiones para introducir los datos. Debido a que es responsabilidad del diseñador del sistema planear las contingencias y el error humano, también deben tomarse provisiones para cambiar datos incorrectos después que se ha hecho una entrada y para eliminar por completo un valor introducido con anterioridad. También pueden hacerse subdivisiones similares para el módulo de reportes. La figura 1.9 ilustra un diagrama de estructura de segundo nivel para un sistema de seguimiento de inventario que incluye estos refinamientos adicionales.

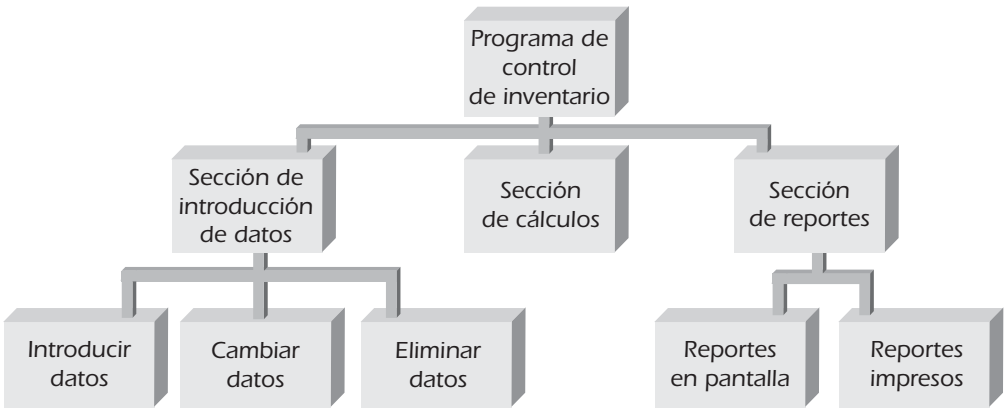


Figura 1.9 *Diagrama de una estructura refinada de segundo nivel.*

El proceso de refinar una solución continúa hasta que el requerimiento más pequeño se ha incluido dentro de la solución. Note que el diseño produce una estructura en forma de árbol, donde los niveles se ramifican conforme pasamos de la parte superior de la estructura a la parte inferior. Cuando el diseño está completo cada tarea designada en un cuadro es codificada, por lo general, en conjuntos separados de instrucciones que se ejecutan cuando son llamadas por tareas superiores en la estructura.

Paso 3 Codificar la solución

Este paso, el cual también se conoce como escribir el programa y poner en práctica la solución, consiste en traducir la solución de diseño elegida en un programa de computadora. Si los pasos de análisis y solución se han realizado en forma correcta, el paso de codificación se vuelve bastante mecánico. En un programa bien diseñado, los planteamientos que forman el programa se conformarán, sin embargo, con ciertos patrones o estructuras bien definidos en el paso de solución. Estas estructuras controlan la forma en que el programa se ejecuta y consiste en los siguientes tipos:

1. Secuencia
2. Selección
3. Iteración
4. Invocación

La **secuencia** define el orden en que son ejecutadas las instrucciones por el programa. La especificación de cuál instrucción entra primero, cuál en segundo lugar, etc., es esencial si el programa ha de lograr un propósito bien definido.

La **selección** proporciona la capacidad para hacer una elección entre diferentes operaciones, dependiendo del resultado de alguna condición. Por ejemplo, el valor de un número puede comprobarse antes que una divisións sea realizada. Si el número no es cero, puede usarse como el denominador de una operación de división; de lo contrario, la división no se ejecutará y se mostrará al usuario un mensaje de advertencia.

La **iteración**, la cual también se denomina *bucle*, *ciclo* o *repetición*, proporciona la capacidad para que la misma operación se repita con base en el valor de una condición. Por ejemplo, podrían introducirse y sumarse grados de manera repetida hasta que un grado negativo sea introducido. Ésta sería la condición que significa el fin de la entrada y adición repetitiva de grados. En ese punto podría ejecutarse el cálculo de un promedio para todos los grados introducidos.

La **invocación** implica invocar, o solicitar, un conjunto de instrucciones cuando sea necesario. Por ejemplo, el cálculo del pago neto de una persona implica las tareas de obtener las tarifas de salario y las horas trabajadas, calcular el pago neto y proporcionar un reporte o cheque por la cantidad requerida. Por lo general una de estas tareas individuales se codificarían como unidades separadas que son llamadas a ejecución, o invocadas, según se necesiten.

Paso 4 Probar y corregir el programa

El propósito de probar es verificar que el programa funciona en forma correcta y en realidad cumple con sus requerimientos. En teoría, las pruebas revelarían todos los errores del programa. (En terminología de computación, un error de programa se conoce como **bug**.[4]) En la práctica, esto requeriría comprobar todas las combinaciones posibles de ejecución de las instrucciones. Debido al tiempo y al esfuerzo requeridos, esto por lo general es una meta imposible, excepto para programas simples en extremo. (En la sección 4.8 se ilustra por qué por lo general ésta es una meta imposible.)

[4]La derivación de este término es bastante interesante. Cuando un programa dejó de ejecutarse en la Mark I, en la Universidad de Harvard, en septiembre de 1945, Grace Hopper rastreó el mal funcionamiento hasta llega a un insecto muerto que había entrado en los circuitos eléctricos. Registró el incidente en su bitácora a las 15:45 horas como "Interruptor #70. . . (polilla) en el interruptor. Primer caso real de bug (insecto) encontrado".

Debido a que no es posible realizar pruebas exhaustivas para la mayor parte de los problemas, han evolucionado diferentes filosofías y métodos de prueba. En su nivel más básico, sin embargo, la prueba requiere de un esfuerzo consciente para asegurarse que un programa funciona en forma correcta y produce resultados significativos. Esto quiere decir que debe meditarse con cuidado lo que se pretende lograr con la prueba y los datos que se usarán en la misma. Si la prueba revela un error (bug), puede iniciarse el proceso de depurar, el cual incluye localizar, corregir y verificar la corrección. Es importante percatarse que *aunque la prueba puede revelar la presencia de un error, no necesariamente indica la ausencia de uno*. Por tanto, *el hecho que una prueba revele un error no indica que otro no esté al acecho en algún otro lugar del programa*.

Para detectar y corregir errores en un programa es importante desarrollar un conjunto de datos de prueba por medio de los cuales determinar si el programa proporciona respuestas correctas. De hecho, un paso comúnmente aceptado en el desarrollo de software muchas veces incluye planear los procedimientos de prueba y crear datos de prueba significativos antes de escribir el código. Esto ayuda a ser más objetivo respecto a lo que debe hacer el programa debido a que en esencia elude cualquier tentación subconsciente después de codificar datos de prueba que no funcionarán. Los procedimientos para probar un programa deberán examinar toda las situaciónes posibles bajo las que se usará el programa. El programa deberá probarse con datos en un rango razonable, al igual que dentro de los límites y en áreas donde el programa debería indicar al usuario que los datos son inválidos. Desarrollar buenos procedimientos y datos de prueba para problemas complejos puede ser más difícil que escribir el código del programa en sí.

La tabla 1.1. enumera la cantidad relativa de esfuerzo que por lo general se dedica en cada uno de estos cuatro pasos de desarrollo y diseño en proyectos de programación comercial grandes. Como muestra este listado, la codificación no es el mayor esfuerzo en esta fase. Muchos programadores novatos tienen problemas debido a que dedicaron la mayor parte de su tiempo a escribir el programa, sin entender por completo el problema o diseñar una solución apropiada. En este aspecto, vale la pena recordar el proverbio de programación, "*Es imposible escribir un programa exitoso para un problema o aplicación que no se ha comprendido por completo*". Un proverbio equivalente e igual de valioso es "*Entre más pronto se comienza a codificar un programa, por lo general tomará más tiempo completarlo*".

Tabla 1.1 Esfuerzo dedicado a la fase I

Paso	Esfuerzo
Analizar el problema	10%
Desarrollar una solución	20%
Codificar la solución	20%
Probar el programa	50%

Fase II. Documentación

Una gran cantidad de trabajo se vuelve inútil o se pierde y deben repetirse demasiadas tareas debido a una documentación inadecuada, por lo que se puede concluir que documentar el trabajo es uno de los pasos más importantes en la solución de problemas. En realidad, durante los pasos de análisis, diseño, codificación y prueba se crean muchos documentos esen-

ciales. Completar la documentación requiere recopilar estos documentos, agregar material práctico para el usuario y presentarlo en una forma que sea de la mayor utilidad.

Aunque no es unánime la clasificación, en esencia existen cinco documentos para toda solución de problema:

1. Descripción del programa
2. Desarrollo y cambios del algoritmo
3. Listado del programa bien comentado
4. Muestras de las pruebas efectuadas
5. Manual del usuario

"Ponerse en los zapatos" de un integrante del equipo de una empresa grande que podría ser el usuario de su trabajo, desde la secretaria hasta el programador, analistas y la gerencia, debería ayudarle a hacer claro el contenido de la documentación importante. La fase de documentación comienza de manera formal en la fase de desarrollo y diseño y continúa hasta la fase de mantenimiento.

Fase III. Mantenimiento

Esta fase tiene que ver con la corrección continua de problemas, revisiones para satisfacer necesidades cambiantes y la adición de características nuevas. El mantenimiento con frecuencia es el esfuerzo mayor, la fuente principal de ingresos y la más duradera de las fases de ingeniería. Mientras el desarrollo puede tomar días o meses, el mantenimiento puede continuar por años o décadas. Entre más completa es la documentación, el mantenimiento podrá efectuarse de manera más eficiente y el cliente y el usuario serán más felices.

Respaldo

Aunque no es parte del proceso de diseño formal, es esencial hacer y conservar copias de respaldo del programa en cada paso del proceso de programación y depuración. Es fácil eliminar o cambiar la versión de trabajo actual de un programa más allá del reconocimiento. Las copias de respaldo permiten la recuperación de la última etapa de trabajo con un esfuerzo mínimo. La versión de trabajo final de un programa útil deberá respaldarse al menos dos veces. A este respecto, otro proverbio de programación útil es "*El respaldo no es importante si no le importa empezar todo de nuevo*".

Muchas empresas conservan al menos un respaldo en el sitio, donde pueda recuperarse con facilidad, y otra copia de respaldo ya sea en una caja fuerte a prueba de fuego o en una ubicación remota.

Ejercicios 1.2

1. **a.** Enumere y describa los cuatro pasos requeridos en la etapa de diseño y desarrollo de un programa.
   b. Además de la etapa de diseño y desarrollo, ¿cuáles son las otras dos etapas requeridas para producir un programa y por qué son necesarias?

2. Una nota de su supervisor, el señor J. Bosworth, dice:

   Solucione nuestros problemas de iluminación.

   —J. Bosworth

   a. ¿Cuál debería ser su primera tarea?
   b. Cómo se llevaría a cabo esta tarea?
   c. ¿Cuánto tiempo espera que tome esta tarea, suponiendo que todos cooperen?

3. El desarrollo del programa es sólo una fase en el procedimiento de desarrollo de software general. Asumiendo que la documentación y el mantenimiento requieren 60% del esfuerzo de software total en el diseño de un sistema, y usando la tabla 1.1, determine la cantidad de esfuerzo requerido para la codificación del programa inicial como un porcentaje del esfuerzo de software total.

4. Muchas personas que solicitan un programa o sistema por primera vez consideran que la codificación es el aspecto más importante del desarrollo del programa. Sienten que saben lo que necesitan y piensan que el programador puede comenzar a codificar con un tiempo mínimo dedicado al análisis. Como programador, ¿qué dificultades puede prever al trabajar en esas condiciones?

5. Muchos usuarios novatos tratan de contratar a los programadores por una cuota fija (la cantidad total que se va a pagar se fija con anticipación). ¿Cuál es la ventaja para el usuario al hacer este arreglo? ¿Cuál es la ventaja para el programador al hacer este arreglo? ¿Cuáles son algunas desventajas tanto para el usuario como para el programador en este arreglo?

6. Muchos programadores prefieren trabajar con una tarifa por hora. ¿Por qué piensa que esto es así? ¿Bajo qué condiciones sería ventajoso para un programador darle a un cliente un precio fijo por el esfuerzo de programación?

7. Los usuarios experimentados por lo general desean una descripción redactada con claridad del trabajo de programación que se hará, incluyendo una descripción completa de lo que hará el programa, fechas de entrega, calendarios de pago y requerimientos de prueba. ¿Cuál es la ventaja para el usuario al requerir esto? ¿Cuál es la ventaja para el programador al trabajar bajo este acuerdo? ¿Qué desventajas tiene este acuerdo tanto para el usuario como para el programador?

1.3 ALGORITMOS

Antes que se escriba un programa, el programador debe entender con claridad qué datos van a usarse, el resultado deseado y el procedimiento que va a utilizarse para producir este resultado. El procedimiento, o solución, seleccionado se conoce como algoritmo. Con más precisión, un **algoritmo** se define como una secuencia paso a paso de instrucciones que deben realizarse y describe cómo han de procesarse los datos para producir las salidas deseadas. En esencia, un algoritmo responde la pregunta: "¿Qué método se usará para resolver este problema?".

Sólo después de entender con claridad los datos que se usarán y seleccionar un algoritmo (los pasos específicos requeridos para producir el resultado deseado) podemos codificar

el programa. Vista bajo esta luz, la programación es la traducción de un algoritmo seleccionado a un lenguaje que pueda usar la computadora.

Para ilustrar un algoritmo, se considerará un problema simple. Suponga que un programa debe calcular la suma de todos los números enteros del 1 al 100. La figura 1.10 ilustra tres métodos que podrían usarse para encontrar la suma requerida. Cada método constituye un algoritmo.

Es evidente que la mayoría de las personas no se molestaría en enumerar las posibles alternativas en una manera paso por paso detallada, como lo hemos hecho aquí, y luego seleccionar uno de los algoritmos para solucionar el problema. Pero claro, la mayoría de las personas no piensa en forma algorítmica; tiende a pensar de manera heurística.

Método 1 - Columnas: *Ordenar los números del 1 al 100 en una columna y sumarlos*

```
   1
   2
   3
   4
   .
   .
   .
  98
  99
+100
----
5050
```

Método 2 - Grupos: *Ordenar los números en grupos que sumen 101 y multiplicar el número de grupos por 101*

```
1+100=101
 2+99=101
 3+98=101
 4+97=101   } 50 grupos
   .  .     }        ↓
   .  .     }   (50x101=5050)
49+52=101   }
50+51=101   }
```

Método 3 - Fórmula: *Usar la fórmula*

$$\text{suma} = \frac{n(a+b)}{2}$$

donde

n = número de términos que se van a sumar (100)
a = primer número que será sumado (1)
b = último número que será sumado (100)

$$\text{suma} = \frac{100(1+100)}{2} = 5050$$

Figura 1.10 Sumar los números del 1 al 100.

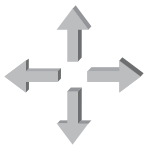
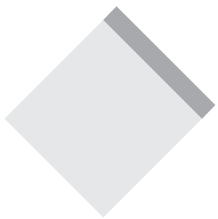
Símbolo	Nombre	Descripción
	Terminal	Indica el principio o fin de un programa
	Entrada/Salida	Indica una operación de entrada o salida
	Proceso	Indica cálculo o manipulación de datos
	Líneas de flujo	Usadas para conectar los otros símbolos del diagrama de flujo e indica el flujo lógico
	Decisión	Indica un punto de ramificación del programa
	Iteración	Indica los valores inicial, límite y de incremento de una iteración
	Proceso predefinido	Indica un proceso predefinido, como llamar a una función
	Conector	Indica una entrada a, o salida de, otra parte de un diagrama de flujo o un punto de conexión
	Reporte	Indica un reporte de salida escrito

Figura 1.11 *Símbolos de diagrama de flujo.*

Por ejemplo, si tuviera que cambiar una llanta desinflada en su automóvil, no pensaría en todos los pasos requeridos, tan sólo cambiaría la llanta o llamaría a alguien que hiciera el trabajo. Éste es un ejemplo de pensamiento heurístico.

Por desgracia, las computadoras no responden a comandos heurísticos. Una instrucción general como "sumar los números del 1 al 100" no significa nada para una computadora porque sólo puede responder a comandos algorítmicos escritos en un lenguaje aceptable como C++. Para programar una computadora con éxito, debe entender con claridad esta diferencia entre comandos algorítmicos y heurísticos. Una computadora es una máquina "que responde a algoritmos"; no es una máquina "que responda a la heurística". No se le puede decir a una computadora que cambie una llanta o sume los números del 1 al 100. En cambio, debe dársele a la computadora un conjunto de instrucciones paso por paso detallado que, de manera colectiva, forma un algoritmo. Por ejemplo, el siguiente conjunto de instrucciones forma un método detallado, o algoritmo, para determinar la suma de los números del 1 al 100:

Establezca que n es igual a 100
Establezca a = 1
Establezca que b es igual a 100
Calcule la suma $= \dfrac{n(a + b)}{2}$
Imprima la suma

Note que estas instrucciones no son un programa de computadora. A diferencia de un programa, el cual debe escribirse en un lenguaje al que pueda responder la computadora, un algoritmo puede escribirse o describirse en varias formas. Cuando se utilizan enunciados en español o en inglés para describir el algoritmo (los pasos de procesamiento), como en este ejemplo, la descripción se llama **seudocódigo**. Cuando se usan ecuaciones matemáticas, la descripción se llama **fórmula**. Cuando se usan diagramas que emplean los símbolos mostrados en la figura 1.11, la descripción se conoce como un **diagrama de flujo**. La figura 1.12 ilustra el uso de estos símbolos para describir un algoritmo para determinar el promedio de tres números.

Debido a que los diagramas de flujo son engorrosos para revisar y pueden soportar con facilidad prácticas de programación poco estructuradas, han perdido el favor de los programadores profesionales, mientras el uso de seudocódigo para expresar la lógica de los algoritmos ha ganado una aceptación creciente. Al describir un algoritmo usando seudocódigo, se usan enunciados cortos en español. Por ejemplo, un seudocódigo aceptable para describir los pasos necesarios para calcular el promedio de tres números es:

Introducir los tres números en la memoria de la computadora
Calcular el promedio sumando los números y dividiendo la suma entre tres
Mostrar el promedio

Sólo después que se ha seleccionado un algoritmo y el programador entiende los pasos requeridos puede escribirse el algoritmo usando instrucciones en lenguaje de computadora. La redacción de un algoritmo usando instrucciones en lenguaje de computadora se llama codificar el algoritmo, lo cual es el tercer paso en nuestro procedimiento de desarrollo del programa (véase la figura 1.13). La mayor parte de la primera parte de este texto está dedicada a mostrarle cómo desarrollar y codificar algoritmos en C++.

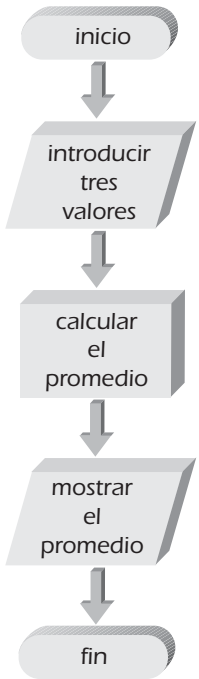


Figura 1.12 Diagrama de flujo para calcular el promedio de tres números.

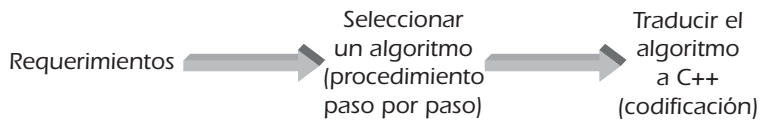


Figura 1.13 Codificación de un algoritmo.

Ejercicios 1.3

1. Determine un procedimiento paso a paso (lista de pasos) para hacer las siguientes tareas. (*Nota: No hay una sola respuesta correcta para cada una de estas tareas. El ejercicio está diseñado para brindarle práctica en convertir comandos tipo heurístico en algoritmos equivalentes y hacer el cambio entre los procesos de pensamiento implicados en los dos tipos de pensamiento.)*
   - **a.** Arreglar una llanta desinflada
   - **b.** Hacer una llamada telefónica
   - **c.** Iniciar sesión en una computadora
   - **d.** Asar un pavo
2. ¿Los procedimientos que desarrolló en el ejercicio 1 son algoritmos? Discuta por qué sí o por qué no.

3. Determine y describa un algoritmo (lista de pasos) para intercambiar los contenidos de dos tazas de líquido. Suponga que dispone de una tercera taza para conservar el contenido de cualquier taza de manera temporal. Cada taza deberá enjuagarse antes que cualquier líquido nuevo se vierta en ella.

4. Escriba un conjunto de instrucciones detallado, en español, para calcular la resistencia de los siguientes resistores conectados en serie: *n* resistores, cada uno con una resistencia de 56 ohmios, *m* resistores, cada uno con una resistencia de 33 ohmios, y *p* resistores, cada uno con una resistencia de 15 ohmios. Note que la resistencia total de los resistores conectados en serie es la suma de todas las resistencias individuales.

5. Escriba un conjunto de instrucciones detalladas paso a paso, para encontrar el número más pequeño en un grupo de tres números enteros.

6. **a.** Escriba un conjunto de instrucciones detalladas paso a paso, para calcular el número menor de billetes en dólares necesarios para pagar una factura de una cantidad TOTAL. Por ejemplo, si TOTAL fuera $97, los billetes consistirían en uno de $50, dos de $20, uno de $5 y dos de $1. (Para este ejercicio, suponga que sólo están disponibles billetes de $100, $50, $20, $10, $5 y $1.)
   b. Repita el ejercicio 6a, pero suponga que la factura debe pagarse sólo con billetes de $1.

7. **a.** Escriba un algoritmo para localizar la primera ocurrencia del nombre JEANS en una lista de nombres ordenada al azar.
   b. Discuta cómo podría mejorar su algoritmo para el ejercicio 7a si la lista de nombres estuviera en orden alfabético.

8. Escriba un algoritmo para determinar las ocurrencias totales de la letra *e* en cualquier enunciado.

9. Determine y escriba un algoritmo para clasificar cuatro números en orden ascendente (de menor a mayor).

1.4 ERRORES COMUNES DE PROGRAMACIÓN

Los errores más comunes asociados con el material presentado en este capítulo son los siguientes:

1. Un error de programación importante cometido por la mayoría de los programadores principiantes es apresurarse a escribir y correr un programa antes de entender por completo qué se requiere, incluyendo los algoritmos que se usarán para producir el resultado deseado. Un síntoma de esta prisa por introducir un programa en la computadora es la falta de cualquier documentación, o incluso un bosquejo de un programa o programa escrito en sí. Pueden detectarse muchos problemas con sólo revisar una copia del programa o incluso una descripción del algoritmo escrito en seudocódigo.

2. Un segundo error importante es no respaldar un programa. Casi todos los programadores nuevos cometen este error hasta que pierden un programa que les ha tomado un tiempo considerable codificar.

3. El tercer error cometido por muchos programadores novatos es la falta de comprensión de que las computadoras sólo responden a algoritmos definidos de manera explícita. Pedirle a una computadora que sume un grupo de números es bastante diferente que decirle a un amigo que sume los números. A la computadora deben dársele las instrucciones precisas para hacer la adición en un lenguaje de programación.

1.5 RESUMEN DEL CAPÍTULO

1. Los programas usados para operar una computadora se denominan *software*.

2. Los lenguajes de programación se presentan en una variedad de formas y tipos. Los programas en *lenguaje de máquina*, también conocidos como *programas ejecutables*, contienen los códigos binarios que pueden ser ejecutados por una computadora. Los *lenguajes ensambladores* permiten el uso de nombres simbólicos para operaciones
matemáticas y direcciones de memoria. Los programas escritos en lenguajes ensambladores deben ser convertidos en lenguajes de máquina, usando programas traductores llamados *ensambladores*, antes que los programas puedan ser ejecutados. Los lenguajes ensambladores y de máquina se denominan *lenguajes de nivel bajo*.

Los *lenguajes compilados* e *interpretados* se denominan *lenguajes de alto nivel*. Esto significa que son escritos usando instrucciones que se parecen a un lenguaje escrito, como el inglés, y pueden ejecutarse en una variedad de tipos de computadora. Los lenguajes compilados requieren un *compilador* para traducir el programa en una forma de lenguaje binario, mientras los lenguajes interpretados requieren un *intérprete* para hacer la traducción.

3. Como una disciplina, la *ingeniería de software* se ocupa de crear programas y sistemas legibles, eficientes, confiables y mantenibles.

4. El procedimiento de desarrollo de software consta de tres fases:

- Desarrollo y diseño del programa
- Documentación
- Mantenimiento

5. La fase de desarrollo y diseño del programa consta de cuatro pasos bien definidos:

- Analizar el problema
- Desarrollar una solución
- Codificar la solución
- Prueba y corrección de la solución

6. Un *algoritmo* es un procedimiento paso por paso que deben realizarse y describe cómo ha de ejecutarse un cálculo o tarea.

7. Un *programa de computadora* es una unidad independiente de instrucciones y datos usados para operar una computadora y producir un resultado específico.
8. Las cuatro estructuras de control fundamentales usadas en la codificación de un algoritmo son
   - Secuencia
   - Selección
   - Iteración
   - Invocación

1.6 APÉNDICE DEL CAPÍTULO: HARDWARE DE COMPUTACIÓN Y CONCEPTOS DE ALMACENAMIENTO

Todas las computadoras, desde las grandes supercomputadoras que cuestan millones de dólares hasta las computadoras personales de escritorio más pequeñas, deben realizar un conjunto mínimo de funciones y proporcionar la capacidad para:

1. Aceptar entradas
2. Mostrar salidas
3. Almacenar información en un formato lógico consistente (tradicionalmente binario)
4. Ejecutar operaciones aritméticas y lógicas en los datos de entrada o en los almacenados
5. Supervisar, controlar y dirigir la operación y secuencia general del sistema

La figura 1.14 ilustra los componentes de la computadora que respaldan estas capacidades y que de manera colectiva forman el **hardware** de la computadora.

La **unidad de aritmética y lógica (ALU)** ejecuta todas las funciones aritméticas y lógicas como adición y sustracción, y las proporcionadas por la computadora.

La **unidad de control** dirige y supervisa la operación general de la computadora. Rastrea el lugar de la memoria donde reside la siguiente instrucción, emite las señales necesarias para leer datos y escribir datos en otras unidades en el sistema y controla la ejecución de todas las instrucciones.

La **unidad de memoria** almacena información en un formato lógico consistente. De manera típica, tanto instrucciones como datos se almacenan en la memoria, por lo general en áreas separadas y distintas.

La **unidad de entrada y salida (I/O o E/S)** proporciona la interfaz a la que se conectan dispositivos periféricos como teclados, monitores, impresoras y lectores de tarjetas.

Almacenamiento secundario: Debido a que la memoria principal en cantidades muy grandes aún es relativamente cara y volátil (lo cual significa que la información se pierde cuando se suspende la energía), no es práctica como un área de almacenamiento permanente para programas y datos. Para este propósito se usan dispositivos de almacenamiento secundario o auxiliar. Aunque los datos se han almacenado en tarjetas perforadas, cinta de papel y otros medios en el pasado, casi todo el almacenamiento secundario se hace ahora en cintas magnéticas, discos magnéticos y CD-ROM.

En las primeras computadoras disponibles en forma comercial en la década de los años 50, todas las unidades de hardware se construían usando relés y tubos catódicos, y el almacenamiento secundario consistía en tarjetas perforadas. Las computadoras resultantes eran piezas de equipo grandes en extremo, capaces de hacer miles de cálculos por segundo que costaban millones de dólares.

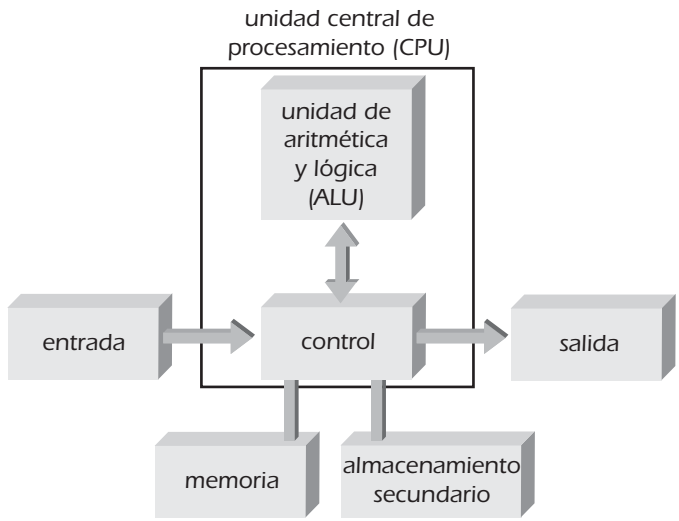


Figura 1.14 *Unidades básicas de hardware de una computadora.*

En la década de los años 60 con la introducción de los transistores, se redujeron tanto el tamaño como el costo del hardware de la computadora. El transistor era aproximadamente una vigésima parte de su contraparte, el tubo catódico. El tamaño pequeño del transistor permitió a los fabricantes combinar la unidad de aritmética y lógica con la unidad de control en una sola unidad. Esta unidad combinada se llama **unidad central de procesamiento (CPU)**. La combinación de la ALU y la unidad de control en una CPU tiene sentido porque la mayoría de las señales de control generadas por un programa están dirigidas a la ALU en respuesta a instrucciones aritméticas y lógicas dentro del programa. Combinar la ALU con la unidad de control simplificó la interfaz entre estas dos unidades y proporcionó una velocidad de procesamiento mejorada.

A mediados de la década de los años 60 se implementó la introducción de circuitos integrados (IC), los cuales produjeron otra reducción significativa en el espacio requerido para producir una CPU. Al principio, los circuitos integrados se fabricaban hasta con 100 transistores en un solo chip de silicio de 1 cm^2. Tales dispositivos se conocen como circuitos integrados de pequeña escala (SSI). Las versiones actuales de estos chips contienen cientos de miles a más de un millón de transistores y se conocen como chips integrados a gran escala (VLSI).

La tecnología de chips VLSI ha proporcionado los medios para transformar las computadoras gigantes de la década de los años 50 en las computadoras personales de escritorio y portátiles de la actualidad. Cada unidad individual requerida para formar una computadora (CPU, memoria e I/O) se fabrica ahora en un chip VLSI individual, y la CPU de un solo chip

se denomina **microprocesador**. La figura 1.15 ilustra cómo se conectan estos chips en forma interna dentro de las computadoras personales actuales, como las PC de IBM.

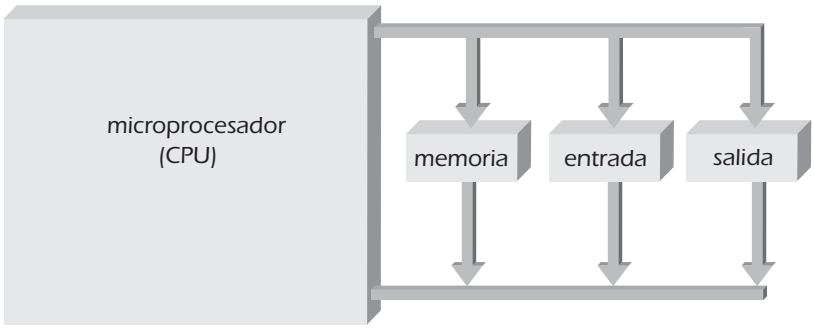


Figura 1.15 Conexiones de chips VLSI para una computadora de escritorio.

En forma concurrente con la reducción notable en el tamaño del hardware de la computadora ha habido una disminución igual de impresionante en el costo y un aumento en las velocidades de procesamiento. Hardware de computadora equivalente que costaba más de un millón de dólares en 1950 ahora puede comprarse por menos de quinientos dólares. Si las mismas reducciones ocurrieran en la industria automotriz, por ejemplo, ¡un Rolls Royce podría comprarse ahora por diez dólares! Las velocidades de procesamiento de las computadoras actuales también se han incrementado por un factor de miles sobre sus predecesores de la década de los años 50, con las velocidades de cómputo de las máquinas actuales midiéndose en millones de instrucciones por segundo (MIPS) y miles de millones de instrucciones por segundo (BIPS).

Almacenamiento de computadora

Los componentes físicos usados en la fabricación de una computadora requieren que los números y letras dentro de su unidad de memoria no se almacenen usando los mismos símbolos que la gente usa. El número 126, por ejemplo, no se almacena usando los números 1, 2 y 6. Ni la letra que reconocemos como A se almacena usando este símbolo. En esta sección veremos por qué es así y cómo almacenan números las computadoras. En el capítulo 2 se verá cómo se almacenan las letras.

La pieza más pequeña y básica de datos en una computadora se llama **bit**. Desde el punto de vista físico, un bit en realidad es un interruptor que puede abrirse o cerrarse. La convención que se seguirá es que las posiciones abierto y cerrado de cada interruptor se representan como un 0 y un 1, respectivamente.[5]

Un solo bit que puede representar los valores 0 y 1, por sí solo, tiene utilidad limitada. Todas las computadoras, por consiguiente, agrupan un número establecido de bits, tanto para su almacenamiento como para su transmisión. El agrupamiento de ocho bits para formar una unidad más grande es un estándar casi universal en la computación. Tales grupos se conocen como **bytes**. Un solo byte consistente en ocho bits, donde cada bit es 0 o 1, puede representar cualquiera de 256 patrones distintos. Éstos consisten del patrón 00000000 (los ocho interruptores abiertos) al patrón 11111111 (los ocho interruptores cerrados) y todas las

[5]Esta convención, por desgracia, es bastante arbitraria, y con frecuencia se encontrará la correspondencia inversa donde las posiciones abierto y cerrado son representadas como 1 y 0, respectivamente.

combinaciones intermedias posibles de 0 y 1. Cada uno de estos patrones puede usarse para representar ya sea una letra del alfabeto, otros caracteres individuales como un signo de dólar, una coma, etc., un solo dígito, o números que contienen más de un dígito. La colección de patrones consistentes en 0 y 1 que se usan para representar letras, dígitos individuales y otros caracteres individuales se llama **código de caracteres**. (Uno de estos códigos, llamado código ASCII, se presenta en la sección 2.3.) Los patrones usados para almacenar números se llaman **código de números**, uno de los cuales se presenta a continuación.

Números en complemento a dos

El código de números más común para almacenar valores enteros dentro de una computadora es la representación de **complemento a dos**. Usando este código, el equivalente entero de cualquier patrón de bits, como 10001101, es fácil de determinar y puede encontrarse para enteros positivos o negativos sin cambio en el método de conversión. Por conveniencia se asumirán patrones de bits del tamaño de un byte consistente en un conjunto de ocho bits cada uno, aunque el procedimiento puede utilizarse en patrones de bits de tamaño más grande.

La forma más fácil de determinar el entero representado por cada patrón de bits es construir primero un recurso simple llamado caja de valores. La figura 1.16 ilustra una de estas cajas para un solo byte. Desde el punto de vista matemático, cada valor en la caja ilustrada en la figura 1.16 representa un aumento en una potencia de dos. Ya que los números complementados a dos deben ser capaces de representar tanto enteros positivos como negativos, la posición en el extremo izquierdo, además de tener la magnitud absoluta más grande, también tiene un signo negativo.

```
-128| 64 | 32 | 16 |  8 |  4 |  2 |  1
----|----|----|----|----|----|----|---
    |    |    |    |    |    |    |
```

Figura 1.16 *Una caja de valores de ocho bits.*

La conversión de cualquier número binario, por ejemplo 10001101, tan sólo requiere insertar el patrón de bits en la caja de valores y sumar los valores que tienen 1 bajo ellos. Por tanto, como se ilustra en la figura 1.17, el patrón de bits 10001101 representa el número entero –115.

```
-128 | 64 | 32 | 16 |  8 |  4 |  2 |  1
 ----|----|----|----|----|----|----|---
   1 |  0 |  0 |  0 |  1 |  1 |  0 |  1
-128 +  0 +  0 +  0 +  8 +  4 +  0 +  1 = -115
```

Figura 1.17 *Conversión de 10001101 a un número en base 10.*

También puede usarse la caja de valores a la inversa para convertir un número entero en base 10 en su patrón de bits binario equivalente. Algunas conversiones, de hecho, pueden hacerse por inspección. Por ejemplo, el número en base 10 –125 se obtiene sumando 3 a –128. Por tanto, la representación binaria de –125 es 10000011, la cual es igual a –128 + 2 + 1. Del mismo modo, la representación en complemento a dos del número 40 es 00101000, la cual es 32 + 8.

Aunque el método de conversión de la caja de valores es engañosamente simple, se relaciona de manera directa con la base matemática subyacente de los números binarios de com-

plemento a dos. El nombre original del código de complemento a dos era código de signo ponderado, el cual se correlaciona en forma directa con la caja de valores. Como implica el nombre **signo ponderado**, cada posición de bit tiene un peso, o valor, de dos elevado a una potencia y un signo. Los signos de todos los bits excepto el bit de la extrema izquierda son positivos y el de la extrema izquierda es negativo.

Al revisar la caja de valores, es evidente que cualquier número binario en complemento a dos con un 1 inicial representa un número negativo, y cualquier patrón de bits con un 0 inicial representa un número positivo. Usando la caja de valores, es fácil determinar los valores más positivos y negativos que pueden almacenarse. El valor más negativo que puede almacenarse en un solo byte es el número decimal –128, el cual tiene el patrón de bits 10000000. Cualquier otro bit diferente de cero tan sólo agregará una cantidad positiva al número. Además, es claro que un número positivo debe tener un 0 como su bit en la extrema izquierda. A partir de esto se puede ver que el número de complemento de dos de ocho bits positivo más grande es 01111111 o 127.

Palabras y direcciones

Uno o más bytes pueden agruparse en unidades más grandes, llamadas **palabras**, lo cual facilita un acceso más rápido y más extenso a los datos. Por ejemplo, recuperar una palabra consistente en cuatro bytes de la memoria de una computadora proporciona más información que la obtenida al recuperar una palabra consistente de un solo byte. Dicha recuperación también es considerablemente más rápida que cuatro recuperaciones de bytes individuales. Sin embargo, este incremento en la velocidad y la capacidad se logra por un aumento en el costo y complejidad de la computadora.

Las primeras computadoras personales, como las máquinas Apple IIe y Commodore, almacenaban y transmitían internamente palabras consistentes de bytes individuales. Las primeras PC de IBM usaban tamaños de palabra consistentes de dos bytes, mientras que las PC más actuales almacenan y procesan palabras consistentes de cuatro bytes cada una.

El número de bytes en una palabra determina los valores máximo y mínimo que pueden ser representados por la palabra. La tabla 1.2 enumera estos valores para palabras de 1, 2 y 4 bytes (cada uno de los valores enumerados puede derivarse usando cajas de valores de 8, 16 y 32 bits, respectivamente).

Tabla 1.2 Valores enteros y tamaño de una palabra

Tamaño de la palabra	Valor del número entero máximo	Valor del número entero mínimo
1 Byte	127	–128
2 Bytes	32 767	–32 768
4 Bytes	2 147 483 647	–2 147 483 648

Además de representar valores enteros, las computadoras también deben almacenar y transmitir números que contienen puntos decimales, los cuales se conocen en matemáticas como números reales. Los códigos usados para números reales, los cuales son más complejos que los usados para enteros, se presentan en el apéndice C.

Consideración de opciones de carrera

Ingeniería aeronáutica y aeroespacial

Entre las más jóvenes de las disciplinas de ingeniería, la ingeniería aeronáutica y aeroespacial se ocupa de todos los aspectos del diseño, producción, prueba y utilización de vehículos o dispositivos que vuelan en el aire (aeronáutica) o en el espacio (aeroespacial), desde alas delta hasta transbordadores espaciales. Debido a que los principios científicos y de ingeniería implicados son tan amplios, los aeroingenieros por lo general se especializan en una subárea que puede superponerse con otros campos de la ingeniería como la ingeniería mecánica, metalúrgica o de materiales, química, civil o eléctrica. Estas subáreas incluyen las siguientes:

1. Aerodinámica. Estudia las características de vuelo de varias estructuras o configuraciones. Las consideraciones típicas son el arrastre y elevación asociados con el diseño de aviones o la aparición del flujo turbulento. Es esencial tener un conocimiento de la dinámica de fluidos. El modelamiento y prueba de todas las formas de aeronaves es parte de esta disciplina.
2. Diseño estructural. El diseño, producción y prueba de aeronaves y naves espaciales para resistir la amplia gama de demandas de vuelo de estos vehículos, así como naves submarinas, son el territorio del ingeniero estructural.
3. Sistemas de propulsión. El diseño de motores de combustión interna, a reacción, y de combustible líquido y sólido para cohetes y su coordinación en el diseño general del vehículo. Los motores de cohete, en especial, requieren ingeniería innovadora para adecuarse a las temperaturas extremas de almacenamiento, mezcla e ignición de combustibles como el oxígeno líquido.
4. Instrumentación y conducción. La industria aeroespacial ha sido líder en el desarrollo y utilización de electrónica de estado sólido en forma de microprocesadores para vigilar y ajustar las operaciones de cientos de funciones de aviones y naves espaciales. Este campo usa la pericia de ingenieros eléctricos y aeroingenieros.
5. Navegación. El cálculo de órbitas dentro y fuera de la atmósfera, y la determinación de la orientación de un vehículo con respecto a puntos en la tierra o en el espacio.

Parte uno

Programación orientada a procedimientos en C++

Capítulos

2 Solución de problemas mediante C++

3 Asignación, formateo y entrada interactiva

4 Estructuras de selección

5 Instrucciones de repetición

6 Modularidad con el uso de funciones

7 Completar lo básico

Aunque C++ es un lenguaje orientado a objetos, fue desarrollado como una extensión de C, el cual es un lenguaje orientado a procedimientos. Por ello, C++ es un lenguaje híbrido que tiene características tanto orientadas a procedimientos como orientadas a objetos. Debido a esta naturaleza híbrida, no sólo es posible escribir un programa C++ completo usando sólo código de procedimientos, es imposible escribir un programa orientado a objetos en C++ que no incluya elementos de procedimientos. Por tanto, un inicio apropiado para aprender C++ requiere familiaridad con sus aspectos de procedimientos.

CAPÍTULO 2
Solución de problemas mediante C++

TEMAS

2.1 INTRODUCCIÓN A C++
LA FUNCIÓN `main()`
EL OBJETO `cout`

2.2 ESTILO DE PROGRAMACIÓN
COMENTARIOS

2.3 TIPOS DE DATOS
TIPOS DE DATOS ENTEROS
DETERMINACIÓN DEL TAMAÑO DE ALMACENAMIENTO
TIPOS DE PUNTO FLOTANTE
NOTACIÓN EXPONENCIAL

2.4 OPERACIONES ARITMÉTICAS
TIPOS DE EXPRESIÓN
DIVISIÓN DE ENTEROS
NEGACIÓN
PRECEDENCIA DEL OPERADOR Y ASOCIATIVIDAD

2.5 VARIABLES E INSTRUCCIONES DE DECLARACIÓN
INSTRUCCIONES DE DECLARACIÓN
DECLARACIONES MÚLTIPLES
ASIGNACIÓN DE MEMORIA
DESPLIEGUE DE LA DIRECCIÓN DE UNA VARIABLE

2.6 APLICACIÓN DEL PROCEDIMIENTO DE DESARROLLO DE SOFTWARE
PASO 1: ANALIZAR EL PROBLEMA
PASO 2: DESARROLLAR UNA SOLUCIÓN
PASO 3: CODIFICAR LA SOLUCIÓN
PASO 4: PRUEBA Y CORRECCIÓN DEL PROGRAMA

2.7 APLICACIONES
APLICACIÓN 1: TRAMPAS DE UN RADAR DE VELOCIDAD
APLICACIÓN 2: REDES DE CONMUTACIÓN TELEFÓNICA

2.8 ERRORES COMUNES DE PROGRAMACIÓN

2.9 RESUMEN DEL CAPÍTULO
CONSIDERACIÓN DE LAS OPCIONES DE CARRERA: INGENIERÍA ELÉCTRICA

2.1 INTRODUCCIÓN A C++

Un programa bien diseñado se construye utilizando una filosofía similar a la usada para edificar un edificio bien diseñado: no sólo sucede; depende de una planeación y ejecución cuidadosa si el diseño final ha de cumplir su propósito pretendido. Del mismo modo en que una parte integral del diseño de un edificio es su estructura, lo mismo sucede con un programa.

Los programas cuyas estructuras consisten de segmentos interrelacionados, organizados en un orden lógico y fácilmente comprensible para formar una unidad integrada y completa, se conocen como **programas modulares** (figura 2.1). Los programas modulares son notoriamente más fáciles de desarrollar, corregir y modificar que los programas construidos de alguna otra manera. En terminología de programación, los segmentos más pequeños usados para construir un programa modular se conocen como **módulos**.

Cada módulo está diseñado y desarrollado para realizar una tarea específica y en realidad es un subprograma pequeño en sí mismo. Un programa C++ completo se construye combinando tantos módulos como sea necesario para producir el resultado deseado. La ventaja de la construcción modular es que el diseño general del programa puede desarrollarse antes de escribir cualquier módulo individual. Una vez que se finalizan los requerimientos para cada módulo, los módulos pueden programarse e integrarse dentro del programa general conforme se completan.

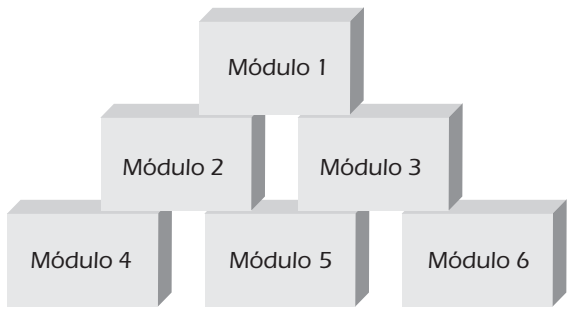


Figura 2.1 *Un programa bien diseñado se construye usando módulos.*

En C++, los módulos pueden ser clases o funciones. Ayuda pensar en una **función** como una máquina pequeña que transforma los datos que recibe en un producto terminado. Por ejemplo, la figura 2.2 ilustra una función que acepta dos números como entradas y multiplica los dos números para producir una salida. Como se muestra, la interfaz para la función son sus entradas y resultados. El proceso de convertir las entradas en resultados se encapsula y oculta dentro de la función. A este respecto, la función puede considerarse como una sola unidad que proporciona una operación de propósito especial. Una analogía similar es apropiada para una **clase**.

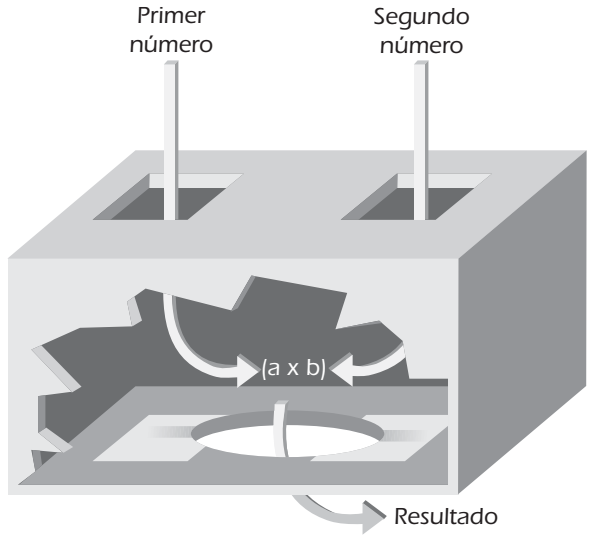


Figura 2.2 *Una función multiplicadora.*

Una **clase** es una unidad más complicada que una función, debido a que contiene tanto datos como funciones apropiadas para manipular los datos. Por tanto, a diferencia de una función, la cual se usa para encapsular un conjunto de operaciones, una clase encapsula tanto datos como uno o más conjuntos de operaciones. Por ello, cada clase contiene todos los elementos requeridos para la entrada, salida y procesamiento de sus objetos y puede considerarse como una pequeña fábrica que contiene materia prima (los datos) y máquinas (las funciones). Al principio, nos interesaremos de manera predominante en el módulo de función más básico.

Un requerimiento importante para diseñar una buena función o clase es darle un nombre que le transmita al lector alguna idea de lo que hace la función o clase. Los nombres admitidos para funciones y clases también se usan para nombrar otros elementos del lenguaje C++ y se denominan de manera colectiva ***identificadores***. Los identificadores pueden formarse por cualquier combinación de letras, dígitos o subrayados (_) seleccionados de acuerdo con las siguientes reglas:

1. El primer carácter del nombre debe ser una letra o subrayado (_).
2. Sólo pueden seguir a la letra inicial, letras, dígitos o subrayados. No se permiten espacios en blanco; se usa el subrayado para separar palabras en un nombre consistente de múltiples palabras o se usan mayúsculas para la primera letra de una o más palabras.
3. El nombre de una función no puede ser una de las palabras clave enumeradas en la tabla 2.1. (Una **palabra clave** es una palabra que es apartada por el lenguaje para un propósito especial y sólo puede usarse en una manera especificada.)[1]
4. El número máximo de caracteres en una función es 1024.[2]

[1]Las palabras clave en C también son palabras reservadas, lo cual significa que deben usarse sólo para el propósito especificado. Intentar usarlas para cualquier otro propósito generará un mensaje de error.

[2]Éste es el mínimo requerido por el estándar ANSI.

Tabla 2.1 Palabras clave

auto	delete	goto	public	this
break	do	if	register	template
case	double	inline	return	typedef
catch	else	int	short	union
char	enum	long	signed	unsigned
class	extern	new	sizeof	virtual
const	float	overload	static	void
continue	for	private	struct	volatile
default	friend	protected	switch	while

Son ejemplos de identificadores válidos de C++:

```
GradARad     intersección    sumarNum    pendiente
bess         mult_dos        HallarMax   densidad
```

Son ejemplos de identificadores inválidos:

1AB3 (empieza con un número, lo cual viola la regla 1)

E*6 (contiene un carácter especial, lo cual viola la regla 2)

while (ésta es una palabra clave, lo cual viola la regla 3)

Además de apegarse a las reglas de identificadores de C++, el nombre de la función siempre debe ser seguido por paréntesis. (La razón para esto se explicará más adelante.) Además, un buen nombre de función deberá ser ***mnemónico***, esto es, una palabra o nombre designado como un auxiliar para la memoria. Por ejemplo, el nombre de función `GradARad()` (note que hemos incluido los paréntesis requeridos después del identificador, lo cual lo señala con claridad como un nombre de función) es un mnemónico si es el nombre de una función que convierte grados a radianes. Aquí, el nombre en sí ayuda a identificar lo que hace la función.

Son ejemplos de nombres de función válidos que no son mnemónicos:

```
fácil()    c3po()     r2d2()     la fuerza()    miguel()
```

Los nombres de función que no son mnemónicos no deberán usarse porque no transmiten información sobre lo que hace la función.

Además, los nombres de función se pueden escribir como una mezcla de letras mayúsculas y minúsculas. Esto se está volviendo cada vez más común en C++, aunque no es absolutamente necesario. Los identificadores escritos por completo en mayúsculas por lo general se reservan para constantes simbólicas, un tema que se cubre en la sección 3.5.

Por añadidura, C++ es un lenguaje ***sensible al uso de mayúsculas y minúsculas***. Esto significa que el compilador distingue entre letras mayúsculas y minúsculas. Por tanto, en C++, los nombres TOTAL, total y TotaL, representan tres nombres distintos.

La función `main()`

Una ventaja distintiva de usar funciones y clases en C++ es que la estructura total del programa en general, y de módulos individuales en particular, puede planearse con anticipación, incluyendo provisiones para probar y verificar la operación de cada módulo. Cada función y clase puede escribirse entonces para cumplir con su objetivo pretendido.

Para facilitar la colocación y ejecución ordenada de los módulos, cada programa C++ debe tener una y sólo una función llamada `main()`. La función `main()` se conoce como una **función controladora**, porque controla, o indica, a los otros módulos la secuencia en la que tienen que ejecutarse (figura 2.3).[3]

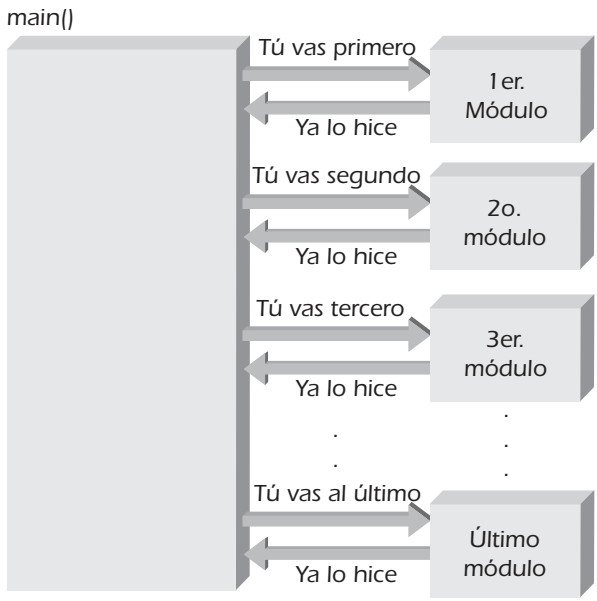


Figura 2.3 *La función* `main()` *dirige a todas las otras funciones.*

La figura 2.4 ilustra una estructura para la función `main()`. La primera línea de la función, en este caso `int main()`, se conoce como ***línea de encabezado de la función***. Una línea de encabezado de la función, la cual siempre es la primera línea de una función, contiene tres fragmentos de información:[4]

1. Qué tipo de dato, si hay alguno, es devuelto por la función.
2. El nombre de la función.
3. Qué tipo de dato, si hay alguno, es enviado a la función.

La palabra clave antes del nombre de la función define el tipo de valor que devuelve la función cuando ha completado su operación. Cuando se coloca antes del nombre de la función,

[3]Los módulos ejecutados a partir de `main()` pueden, a su vez, ejecutar otros módulos. Cada módulo, sin embargo, siempre regresa al módulo que inició su ejecución. Esto se aplica incluso a `main()`, la cual regresa el control al sistema operativo cuando fue iniciado `main()`.

[4]Un método de clase también debe empezar con una línea de encabezado que se apega a estas mismas reglas.

la palabra clave `int` (véase la tabla 2.1) indica que la función devolverá el valor de un número entero. Del mismo modo, cuando los paréntesis que siguen al nombre de la función están vacíos, significa que no se transmitirán datos a la función cuando se esté ejecutando. (Los datos transmitidos a una función al momento de ejecutarla se llaman ***argumentos*** de la función.) Las llaves { y }, determinan el principio y el fin del cuerpo de la función y encierran las instrucciones que la componen. Las instrucciones dentro de las llaves determinan lo que hace la función. Cada instrucción dentro de la función debe terminar con un punto y coma (;).

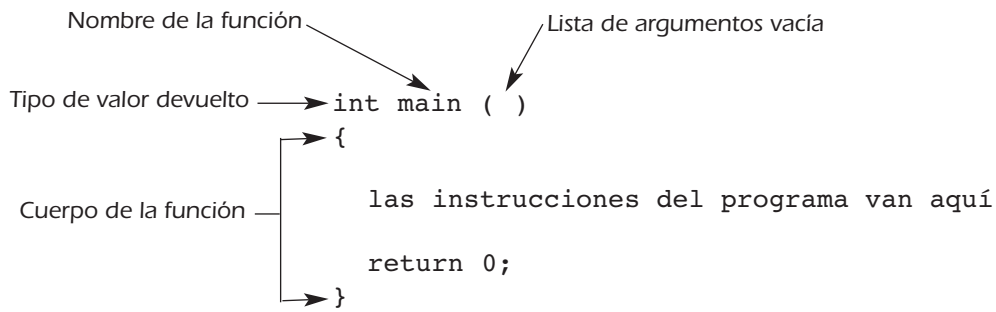


Figura 2.4 *La estructura de una función* `main()`.

Se pueden nombrar y escribir muchas de las funciones C++ personalizadas. De hecho, el resto de este libro se ocupa de manera principal de las instrucciones requeridas para elaborar funciones útiles y la forma de combinar funciones y datos en clases y programas útiles. Cada programa, sin embargo, debe tener una y sólo una función `main()`. Hasta que aprenda cómo pasar y regresar los datos de una función (los temas del capítulo 6), la línea de encabezado ilustrada en la figura 2.4 servirá para todos los programas que sea necesario escribir. Para programas simples, hay que considerar que las primeras dos líneas (que se explican con más detalle en el capítulo 6):

```
int main()
{
```

tan sólo señalan que "el programa comienza aquí", mientras las últimas dos líneas

```
  return 0;
}
```

señalan el final del programa. Por suerte, muchas funciones y clases útiles ya se presentan escritas. Ahora se verá cómo usar un objeto creado a partir de una de estas clases para crear nuestro primer programa funcional en C++.

El objeto `cout`

Uno de los objetos más versátiles y más usados en C++ se llama `cout` (pronunciado "si out"). Este objeto, cuyo nombre se deriva de Console OUTput, es un objeto de salida que envía datos introducidos en él al dispositivo estándar de salida.[5] Para la mayoría de los sis-

[5] El objeto `cout` se crea de manera formal a partir de la clase `ostream`, la cual se describe con más detalle en el capítulo 7.

temas, este dispositivo de salida es una pantalla de video. El objeto `cout` muestra en el monitor cualquier cosa enviada a él. Por ejemplo, si se envían los datos `¡Hola mundo!` a `cout`, estos datos se imprimen (o despliegan) en la pantalla de la terminal. Los datos `¡Hola mundo!` se envían al objeto `cout` encerrando el texto entre comillas, `"poner cadena aquí"`, y colocando el símbolo de inserción `<<` ("enviar a"), antes del mensaje y después del nombre del objeto, como se muestra en la figura 2.5.

```
cout << "¡Hola mundo!";
```

Figura 2.5 *Envío de un mensaje a* `cout`.

Ahora juntemos todo esto en un programa C++ funcional que pueda ejecutarse en su computadora. Considere el programa 2.1.

Programa 2.1

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Hola mundo!";

  return 0;
}
```

La primera línea del programa,

```
#include <iostream>
```

es un comando preprocesador que utiliza la palabra reservada `include`. Los comandos preprocesadores comienzan con un signo de número (#) y ejecutan alguna acción antes que el compilador traduzca el programa fuente a código de máquina. De manera específica, el comando preprocesador `#include` causa que el contenido del archivo invocado, en este caso el archivo `iostream`, sea insertado en donde el comando `#include` aparezca en el programa. El `iostream` es una parte de la biblioteca estándar que contiene, entre otro código, dos clases nombradas `istream` y `ostream`. Estas dos clases proporcionan las declaraciones de datos y métodos utilizados para la entrada y salida de datos, respectivamente. El archivo `iostream` se conoce como un **archivo de encabezado** debido a que siempre se coloca una referencia a él en la parte superior, o cabeza, de un programa C++ usando el comando #include. Puede ser que se pregunte qué tiene que ver el archivo `iostream` con este programa simple. La respuesta es que el objeto `cout` se crea a partir de la clase `ostream`. Por tanto, el archivo de encabezado `iostream` debe incluirse en todos los programas que usen `cout`. Como se indicó en el programa 2.1, los comandos preprocesadores no terminan con un punto y coma.

Punto de Información

¿Qué es la sintaxis?
La **sintaxis** de un lenguaje de programación es el conjunto de reglas para formular instrucciones gramaticalmente correctas en un lenguaje. En la práctica esto significa que una instrucción en C++ con sintaxis correcta tiene la forma apropiada especificada para el compilador. Como tal, el compilador aceptará la instrucción y no generará un mensaje de error.

Debe señalarse que una instrucción o programa individual puede ser correcto desde el punto de vista sintáctico, pero incorrecto desde el punto de vista lógico. Una instrucción o programa así estaría estructurado de manera correcta pero produciría un resultado incorrecto. Esto es similar a un enunciado en español que es correcto desde el punto de vista gramatical pero no tiene sentido. Por ejemplo, aunque el enunciado "El árbol es un gato andrajoso" es gramaticalmente correcto, no tiene sentido.

Después del comando preprocesador `include` hay una instrucción que contiene la palabra reservada `using`. La instrucción,

```
using namespace std;
```

le dice al compilador dónde buscar para encontrar los archivos de encabezado en ausencia de cualquier designación explícita adicional. Se puede pensar en un espacio de nombres como un archivo de código fuente al que tiene acceso el compilador cuando busca clases o funciones preescritas. Debido a que el archivo de encabezado `iostream` está contenido dentro de un espacio de nombres llamado `std`, el compilador usará de manera automática el objeto `cout` de `iostream` a partir de este espacio de nombres siempre que se haga referencia a `cout`. El uso de espacio de nombres le permite crear sus propias clases y funciones con los mismos nombres provistos por la biblioteca estándar y colocarlos en espacios de nombres invocados de manera diferente. Puede señalar al programa cuál clase o función usar al indicar el espacio de nombres donde desea que el compilador busque la clase o función.

La instrucción `using` es seguida por el inicio de la función `main()` del programa. Esta función comienza con la línea de encabezado desarrollada al principio de esta sección. El cuerpo de la función, entre llaves, consta de sólo dos instrucciones. La primera instrucción en `main()` pasa un mensaje al objeto `cout`. El mensaje es la cadena `"¡Hola mundo!"`.

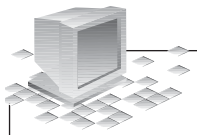
Debido a que `cout` es un objeto de una clase preescrita, no tenemos que escribirlo; está disponible para su uso con sólo activarlo en forma correcta. Como todos los objetos de C++, `cout` sólo puede ejecutar ciertas acciones bien definidas. Para `cout`, la acción es ensamblar datos para mostrar la salida. Cuando se pasa una cadena de caracteres a `cout`, el objeto asegura que la cadena se despliegue de manera correcta en su monitor, como se muestra en la figura 2.6.

```
¡Hola mundo!
```

Figura 2.6 *La salida del programa 2.1.*

Las cadenas en C++ son cualquier combinación de letras, números y caracteres especiales encerrados entre comillas ("colocar cadena aquí"). Las comillas se usan para delimitar (marcar) el comienzo y el final de la cadena y no se consideran parte de la cadena. Por tanto, la cadena de caracteres que forma el mensaje enviado a `cout` debe encerrarse entre comillas, como se ha hecho en el programa 2.1.

Escribiremos otro programa para ilustrar la versatilidad de `cout`. Lea el programa 2.2 para determinar qué hace.



Programa 2.2

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Computadoras, computadoras por todos lados";
  cout << "\n   tan lejos como pueda llegar C";

  return 0;
}
```

Cuando se ejecuta el programa 2.2 se despliega lo siguiente:

```
Computadoras, computadoras por todos lados
   tan lejos como pueda llegar C
```

Tal vez se pregunte por qué no apareció `\n` en la salida. Los dos caracteres \ y n, cuando se usan juntos, se llaman secuencia de escape para una línea nueva. Le indican a `cout` que envíe instrucciones al dispositivo de salida para iniciar una línea nueva. En C++, el carácter de diagonal inversa (\) proporciona un "escape" de la interpretación normal del carácter que la sigue, alterando el significado del siguiente carácter. Si se omitiera la diagonal inversa de la segunda instrucción `cout` en el programa 2.2, la n se imprimiría como la letra n y el programa desplegaría:

```
Computadoras, computadoras por todos lados   tan lejos como pue-
da llegar C
```

Pueden colocarse secuencias de escape para una línea nueva en cualquier parte dentro del mensaje que se transmite a `cout`. Vea si puede determinar el despliegue producido por el programa 2.3.

Programa 2.3

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Computadoras por todos lados\n tan lejos como\n\npueda llegar C";

  return 0;
}
```

La salida para el programa 2.3 es:

```
Computadoras por todos lados tan lejos como
 puede llegar C

puedo ver
```

Ejercicios 2.1

1. Indique si los siguientes son nombres de función válidos. Si son válidos, indique si son nombres mnemónicos. (Recuerde que un nombre de función mnemónico transmite alguna idea sobre el propósito de la función.) Si son nombres inválidos, indique por qué.

```
poder     densidad   m1234$          ampnuevo   1234    abcd
total     tangente   valAbs          computado  b34a    34ab
voltios$  a2B3       while           valMin     seno    $seno
coseno    velocidad  distancianeta   suma       return  pila
```

2. Suponga que se han escrito las siguientes funciones:

   `obtenerLargo(), obtenerAncho(), calcArea(), mostrarArea()`

   a. A partir de los nombres de las funciones, ¿qué piensa que podría hacer cada función?
   b. ¿En qué orden piensa que una función `main()` podría ejecutar estas funciones (con base en sus nombres)?

3. Suponga que se han escrito las siguientes funciones:

   `velocidad(), distancia(), aceleración()`

   A partir de los nombres de las funciones, ¿qué piensa que podría hacer cada función?

4. Determine nombres para funciones que hagan lo siguiente:
   a. Encontrar el promedio de un conjunto de números.
   b. Encontrar el área de un rectángulo.

c. Encontrar el valor mínimo en un conjunto de números.
d. Encontrar la densidad de una puerta de acero.
e. Clasificar un conjunto de números de menor a mayor.

5. Del mismo modo en que la palabra clave `int` se usa para indicar que una función devolverá un número entero, las palabras clave `void`, `char`, `float` y `double` se usan para indicar que una función no devolverá ningún valor o devolverá un carácter, un número en punto flotante en precisión simple, y un número en precisión doble, respectivamente. Usando esta información, escriba líneas de encabezado para una función `main()` que no recibirá argumentos pero que devolverá:

a. ningún valor
b. un carácter
c. un número de punto flotante en precisión simple
d. un número en precisión doble

6. a. Usando `cout`, escriba un programa en C++ que despliegue su nombre en una línea, su domicilio en una segunda línea, y su ciudad, estado y código postal en una tercera línea.

b. Ejecute el programa que ha escrito para el ejercicio 6a en una computadora. (*NOTA:* Debe entender los procedimientos para introducir y ejecutar un programa C++ en la computadora particular que esté usando.)

7. a. Escriba un programa en C++ para desplegar lo siguiente:

```
La cosecante de un ángulo
   es igual a uno dividido entre
     el seno del ángulo.
```

b. Compile y ejecute el programa que ha escrito para el ejercicio 7a en una computadora.

8. a. ¿Cuántas instrucciones `cout` usaría para desplegar lo siguiente:

```
Grados          Radianes
 0              0.0000
90              1.5708
180             3.1416
270             4.7124
360             6.2832
```

b. ¿Cuál es el número mínimo de instrucciones cout que podrían usarse para imprimir la tabla en el ejercicio 8a?
c. Escriba un programa en C++ completo para producir la salida ilustrada en el ejercicio 8a.
d. Ejecute el programa que ha escrito para el ejercicio 8c en una computadora.

9. En respuesta a una secuencia de escape de línea nueva, `cout` coloca el siguiente carácter desplegado al principio de una línea nueva. Esta colocación del siguiente carácter en realidad representa dos operaciones distintas. ¿Cuáles son?

10. ***a.*** Suponiendo un compilador que no es sensible al uso de mayúsculas y minúsculas, determine cuáles de estos nombres de unidades de programa son equivalentes:

```
PROMEDIO   promedio   MODO     BESSEL     Modo
Total      besseL     TeMp     Densidad   TEMP
densiDAD   MEDIA      total    media      moDO
```

b. Vuelva a hacer el ejercicio 10a suponiendo que el compilador es sensible al uso de mayúsculas y minúsculas.

Ejercicios para estructurar un proyecto

La mayor parte de los proyectos, tanto de programación como de otro tipo, por lo general pueden estructurarse en subtareas o unidades de actividad más pequeñas. Estas subtareas más pequeñas a menudo pueden delegarse a diferentes personas, de modo que cuando todas las tareas se terminan e integran, el proyecto o programa está completo. Para los ejercicios 11 a 16, determine un conjunto de subtareas que, unidas, completen el proyecto. Tome en cuenta que hay muchas soluciones posibles para cada ejercicio. El único requisito es que el conjunto de subtareas seleccionadas, cuando se junten, completen la tarea requerida.

NOTA: El propósito de estos ejercicios es que considere las diferentes formas en que pueden estructurarse tareas complejas. Aunque no hay una solución correcta para estos ejercicios, hay soluciones incorrectas y soluciones que son mejores que otras. Una solución incorrecta es aquella que no especifica por completo la tarea. Una solución es mejor que otra si identifica con más claridad o más facilidad lo que debe hacerse.

11. Se le dio la tarea de cablear e instalar luces en el ático de su casa. Determine un conjunto de subtareas que, unidas, harán que logre esto. (*Sugerencia:* La primera subtarea debería determinar la colocación de las lámparas.)

12. Se le da el trabajo de preparar una comida completa para cinco personas el próximo fin de semana. Determine un conjunto de subtareas que, unidas, harán que puedan lograr esto. (*Sugerencia:* Una subtarea, no necesariamente la primera, debería ser comprar los alimentos.)

13. Usted es un estudiante de segundo año de bachillerato y está planeando ir a una escuela superior para estudiar ingeniería eléctrica después de graduarse. Enumere un conjunto de objetivos principales que debe cumplir para alcanzar esta meta. (*Sugerencia:* Una subtarea es "Determinar los cursos adecuados que deberá tomar".)

14. Se le ha dado el trabajo de cultivar un huerto. Determine un conjunto de subtareas para lograr esto. (*Sugerencia:* Una de las subtareas sería planear la disposición del huerto.)

15. Usted es responsable de planificar y organizar un viaje de la familia a un campamento este verano. Enumere un conjunto de subtareas que, unidas, lo lleven a alcanzar este objetivo con éxito. (*Sugerencia:* Una subtarea sería seleccionar la ubicación del campamento.)

16. ***a.*** Un laboratorio nacional de exámenes médicos desea un sistema de cómputo nuevo para analizar los resultados de sus exámenes. El sistema debe ser capaz de procesar los resultados diarios. Además, el laboratorio requiere recuperar y obtener un informe impreso de todos los resultados que cumplen con ciertos cri-

terios, por ejemplo, todos los resultados obtenidos por un medico particular o todos los resultados obtenidos para los hospitales en un estado particular. Determine tres o cuatro unidades del programa en los que podría tratarse por separado este sistema. (*Sugerencia:* Una unidad del programa posible es "Preparar los resultados diarios" para crear los informes de cada día.)

b. Suponga que alguien introduce datos incorrectos para un resultado de examen particular, lo cual se descubre después que los datos se han introducido y almacenado en el sistema. ¿Qué unidad del programa es necesaria para corregir este problema? Discuta por qué una unidad del programa como ésta podría requerirse o no en la mayor parte de los sistemas.

c. Suponga que existe una unidad del programa que permite a un usuario alterar o cambiar datos que se han introducido y almacenado en forma incorrecta. Discuta la necesidad de incluir un "rastreo de auditoría" que permitiría una reconstrucción posterior de los cambios hechos, cuándo se hicieron y quién los hizo.

2.2 ESTILO DE PROGRAMACIÓN

Los programas en C++ empiezan la ejecución al principio de la función `main()`. Debido a que un programa sólo puede tener un punto de inicio, todo programa en lenguaje C++ debe contener una y sólo una función `main()`. Como hemos visto, todas las instrucciones que forman la función `main()` se incluyen luego dentro de las llaves que siguen al nombre de la función. Aunque la función `main()` debe estar presente en todos los programas C++, éste no requiere que la palabra main, los paréntesis `()` o las llaves `{}` se coloquen en alguna forma particular. La forma usada en la sección anterior,

```
int main()
{
  las instrucciones del programa van aquí;

  return 0;
}
```

se eligió sólo por claridad y facilidad en la lectura del programa. Si una de las instrucciones del programa usa el objeto `cout`, debe incluirse el archivo de encabezado `iostream`, al igual que la instrucción `using namespace std;`. Por ejemplo, la siguiente forma general de una función `main()` también funcionaría:

```
int main
 (
 ) { primera instrucción; segunda instrucción;
          tercera instrucción; cuarta
instrucción;
return 0;}
```

Note que puede ponerse más de una instrucción en una línea, o una instrucción puede escribirse en más de una línea. Con excepción de las cadenas, comillas, identificadores y palabras clave, C++ ignora todo el espacio en blanco. (El espacio en blanco se refiere a cualquier combinación de uno o más espacios en blanco, tabuladores o líneas nuevas.) Por

ejemplo, cambiar el espacio en blanco en el programa 2.1 y asegurarse que no se divide la cadena ¡Hola mundo! en dos líneas produce el siguiente programa válido:

```
#include <iostream>
using namespace std;

int main
(
){
cout <<
"¡Hola mundo!";
 return 0;
}
```

Aunque esta versión de `main()` funciona, es un ejemplo de un estilo de programación pobre en extremo. Es difícil de leer y entender. Para legibilidad, la función `main()` siempre deberá escribirse en forma estándar como:

```
int main()
{
  las instrucciones del programa van aquí;

  return 0;
}
```

En esta forma estándar el nombre de la función empieza en la columna 1 y se coloca con los paréntesis requeridos en una sola línea. La llave de apertura del cuerpo de la función sigue en la siguiente línea y se coloca bajo la primera letra de la línea que contiene el nombre de la función. Del mismo modo, la llave que cierra la función se coloca sola en la columna 1 como la última línea de la función. Esta estructura sirve para resaltar la función como una unidad independiente.

Dentro de la función en sí, todas las instrucciones del programa tienen una sangría de por lo menos dos espacios. La sangría es otra buena práctica de programación, en especial si se usa la misma sangría para grupos similares de instrucciones. Revise el programa 2.2 para ver que se usó la misma sangría para ambas llamadas al objeto `cout`.

Mientras progresa en su comprensión y dominio de C++, desarrollará sus propios estándares para las sangrías. Sólo tenga en cuenta que la forma final de sus programas deberán ser consistentes y servir siempre como un auxiliar para la lectura y comprensión de sus programas.

Comentarios

Los comentarios son observaciones explicativas que se hacen dentro de un programa. Cuando se usan con cuidado, los comentarios pueden ser muy útiles para aclarar de qué trata el programa completo, qué pretende lograr un grupo específico de instrucciones o qué pretende hacer una línea. C++ acepta dos tipos de comentarios: de línea y de bloque. Ambos tipos de comentarios pueden colocarse en cualquier parte dentro de un programa y no tienen efecto

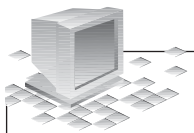
en la ejecución del mismo. El compilador ignora todos los comentarios; están ahí estrictamente para la conveniencia de cualquiera que lea el programa.

Un **comentario de línea** empieza con dos diagonales (`//`) y continúa hasta el final de la línea. Por ejemplo, las siguientes líneas son comentarios de línea:

```
// éste es un comentario
// este programa imprime un mensaje
// este programa calcula una raíz cuadrada
```

Los símbolos `//`, sin espacio en blanco entre ellos, designan el inicio del comentario de línea. El final de la línea en la que está escrito el comentario designa el final del comentario.

Un comentario de línea puede escribirse ya sea en una sola línea o al final de la misma línea que contiene una instrucción del programa. El programa 2.4 ilustra el uso de comentarios de línea dentro de un programa.



Programa 2.4

```
// este programa despliega un mensaje
#include <iostream>
using namespace std;

int main()
{
  cout << "¡Hola mundo!"; // esto produce el despliegue

  return 0;
}
```

El primer comentario aparece en una línea aparte al principio del programa y describe lo que hace el programa. Por lo general ésta es una buena ubicación para incluir un comentario breve que describa el propósito del programa. Si se requieren más comentarios, puede colocarse uno por línea. Por tanto, cuando un comentario es demasiado largo para que quepa en una línea, puede separarse en dos o más comentarios de línea, con cada comentario separado precedido por el símbolo de doble diagonal establecido `//`. El comentario

```
// este comentario es inválido porque
   se extiende en dos líneas
```

producirá un mensaje de error de C++ en su computadora. Este comentario es correcto cuando se escribe así

```
// este comentario se usa para ilustrar un
// comentario que se extiende en dos líneas
```

Sin embargo, los comentarios que abarcan dos o más líneas se escriben de manera más conveniente como comentarios en bloque tipo C que como múltiples comentarios de línea. Los **comentarios en bloque** comienzan con los símbolos `/*` y terminan con los símbolos `*/`. Por ejemplo,

```
/* Éste es un comentario en bloque que
    abarca
    tres líneas */
```

En C++, la estructura del programa pretende que sea legible y comprensible, haciendo innecesario el uso de comentarios extensos. Esto se refuerza si los nombres de función, clase y variable, que se describen en el siguiente capítulo, son seleccionados con cuidado para transmitir su significado a cualquiera que lea el programa. Sin embargo, si el propósito de una función, clase o instrucción aún no está claro a partir de su estructura, nombre o contexto, incluya comentarios donde se necesiten aclaraciones. El código oscuro, sin comentarios, es una señal segura de mala programación y se hace presente cuando el programa debe ser sometido a mantenimiento o ser leído por otros. Del mismo modo, los comentarios excesivos también son una señal de mala programación, porque implican que hubo un razonamiento insuficiente para hacer que el código se explicara por sí solo. Por lo general, cualquier programa que escriba deberá comenzar con una serie de comentarios iniciales al programa que incluyan una descripción breve de éste, su nombre y la fecha de la última modificación. Por consideraciones de espacio, y debido a que todos los programas en este texto fueron escritos por el autor, sólo se usarán comentarios iniciales para descripciones breves de los programas cuando no se proporcionen como parte del texto descriptivo acompañante.

Ejercicios 2.2

1. a. ¿Funcionará el siguiente programa?

```
#include <iostream>
using namespace std;
int main() {cout << "¡Hola mundo!"; return 0;}
```

b. ¿Por qué el programa dado en el ejercicio 1a no es un buen programa?

2. Vuelva a escribir los siguientes programas para que se ajusten a una buena práctica de programación y una sintaxis correcta.

a.
```
#include <iostream>
  int main(
  ){
  cout          <<
  "El momento ha llegado"
  ; return 0;}
```

b.
```
#include <iostream>
using namespace std;
  int main
  (    ){cout << "Newark es una ciudad\n";cout <<
  "En Nueva Jersey\n"; cout <<
  "También es una ciudad\n"
  ; cout << "En Delaware\n"
  ; return 0;}
```

c.
```
#include <iostream>
using namespace std;
 int main() {cout << Leer un programa\n";cout <<
 "es mucho más fácil\n"
 ; cout << "si se usa una forma estándar para main\n")
 ; cout
 <<"y cada instrucción se escribe\n";cout
 <<           "en una línea separada\n")
 ; return 0;}
```

d.
```
#include <iostream.h>
using namespace std;
 int main
 (    ){ cout << "Todo programa en C++"
 ; cout
 <<"\ndebe tener una y sólo una"
 ;
 cout << "función main"
 ;
 cout <<
 "\n la secuencia de escape de caracteres")
 ; cout <<
   "\npara una línea nueva puede colocarse en cualquier
      parte"
 ; cout
 <<"\n dentro del mensaje transmitido a cout"
 ; return 0;}
```

3. a. Cuando se usa en un mensaje, el carácter de diagonal inversa altera el significado del carácter que le sigue inmediatamente. Si deseamos imprimir el carácter de diagonal inversa, tendríamos que proporcionan a `cout` una secuencia de escape para que interprete normalmente la diagonal inversa. ¿Qué carácter piensa que se usa para alterar la forma en que se interpreta un solo carácter de diagonal inversa?

b. Usando su respuesta al ejercicio 3a, escriba la secuencia de escape para imprimir una diagonal inversa.

4. a. Una ***señal (token*** en inglés**)** de un lenguaje de computadora es cualquier secuencia de caracteres que, como una unidad, sin caracteres que intervengan o espacios en blanco, tiene un significado único. Usando esta definición de token, determine si las secuencias de escape, nombres de función y las palabras clave enumeradas en la tabla 1.2 son tokens del lenguaje C++.

b. Analice si agregar un espacio en blanco a un mensaje lo altera. Analice si los mensajes pueden considerarse tokens de C++.

c. Utilizando la definición de token que se dio en el ejercicio 4a, determine si la siguiente instrucción es verdadera: "Excepto por las tokens del lenguaje, C++ ignora todos los espacios en blanco."

2.3 TIPOS DE DATOS

El objetivo de todos los programas es procesar datos, sean numéricos, alfabéticos, de audio o de video. Es fundamental para este objetivo la clasificación de los datos en tipos específicos. Por ejemplo, calcular la trayectoria de un cohete requiere operaciones matemáticas con datos numéricos, y alfabetizar una lista de nombres requiere operaciones de comparación con datos basados en caracteres. Además, algunas operaciones no son aplicables a ciertos tipos de datos. Por ejemplo, no tiene sentido sumar nombres. Para impedir que los programadores intenten ejecutar una operación inapropiada, C++ sólo permite que se ejecuten ciertas operaciones con ciertos tipos de datos.

Los tipos de dato permitidos y las operaciones apropiadas definidas para cada tipo se conocen como tipo de dato. De manera formal, un **tipo de dato** se define como un conjunto de valores y un conjunto de operaciones que pueden aplicarse a estos valores. Por ejemplo, el conjunto de todos los números enteros constituye un conjunto de valores, como lo es el conjunto de todos los números reales (números que contienen un punto decimal). Estos dos conjuntos de números, sin embargo, no constituyen un tipo de dato hasta que se incluye un conjunto de operaciones. Estas operaciones, por supuesto, son las operaciones matemáticas y de comparación. La combinación de un conjunto de valores más las operaciones se vuelve un tipo de dato verdadero.

C++ clasifica los tipos de datos en uno de dos agrupamientos fundamentales: tipos de dato de clase y tipos de dato integrados. Un **tipo de dato de clase**, al cual se hace referencia como una clase, para abreviar, es un tipo de dato creado por el programador. Esto significa que el conjunto de valores y operaciones admitidas es definido por un programador, usando código C++.

Un **tipo de dato** integrado es proporcionado como una parte integral del compilador C++ y no requiere código C++ externo. Por tanto, un tipo de dato integrado puede usarse sin recurrir a adiciones de lenguaje complementarias, como las proporcionadas por el archivo de encabezado `iostream` necesario para el objeto `cout`. Los tipos de dato integrados, los cuales también se conocen como tipos **primitivos**, consisten en los tipos numéricos básicos mostrados en la figura 2.7 y las operaciones enumeradas en la tabla 2.2. Como se puede observar en esta tabla, la mayor parte de las operaciones para tipos integrados son proporcionadas como símbolos. Esto contrasta con los tipos de clases, donde la mayor parte de las operaciones son proporcionadas como funciones.

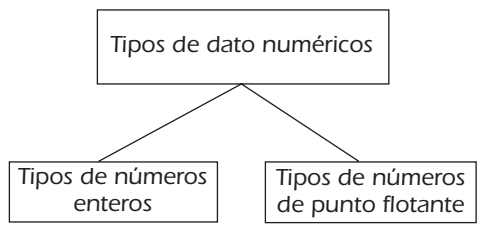


Figura 2.7 Tipos de dato integrados.

Tabla 2.2 Operaciones con tipos de dato integrados

Tipos de dato integrados	Operaciones
`Números enteros`	`+, -, *, /, %, =, ==, !=, <=, >=vsizeof(), y operaciones con bits` (véase el capítulo 15)
`Números de punto flotante`	`+, -, *, /, =, ==, !=, <=, >=, sizeof()`

Al introducir tipos de dato integrados de C++, usaremos literales. Una **literal** es un valor aceptable para un tipo de dato. El término **literal** refleja que dicho valor se identifica de manera explícita a sí mismo. (Otro nombre para una literal es un **valor literal** o **constante**.) Por ejemplo, todos los números, como 2, 3.6 y –8.2, se denominan valores literales porque despliegan sus valores de manera literal. El texto, como `"¡Hola mundo!"` también se denomina valor literal porque el texto se despliega. Usted ha utilizado valores literales a través de su vida y los ha conocido por lo común como números y palabras. En la sección 2.5, se puede observar algunos ejemplos de valores no literales, es decir, valores que no se despliegan a sí mismos sino que son almacenados y se tiene acceso a ellos por medio de identificadores.

Tipos de datos enteros

C++ proporciona nueve tipos de datos integrados, como se muestra en la figura 2.8. La diferencia esencial entre los diversos tipos de datos enteros es la cantidad de almacenamiento usado por cada tipo, el cual afecta al rango de valores que cada tipo es capaz de representar. Los tres tipos más importantes usados en forma casi exclusiva en la mayor parte de las aplicaciones son los tipos de datos `int`, `char` y `bool`. La razón para los tipos restantes es histórica, ya que fueron implementados para acomodar situaciones especiales (un rango de números pequeño o grande). Esto permitía al programador maximizar el uso de memoria al seleccionar un tipo de datos que usara la cantidad más pequeña de memoria consistente con los requerimientos de una aplicación. Cuando las memorias de las computadoras eran pequeñas y costosas en comparación con las actuales, ésta era una preocupación importante. Aunque ya no es una preocupación para la mayor parte de los programas, estos tipos todavía le proporcionan al programador la capacidad para optimizar el uso de memoria cuando es necesario. Por lo general, estas situaciones ocurren en aplicaciones de ingeniería, como en sistemas de control usados en aparatos domésticos y automóviles.

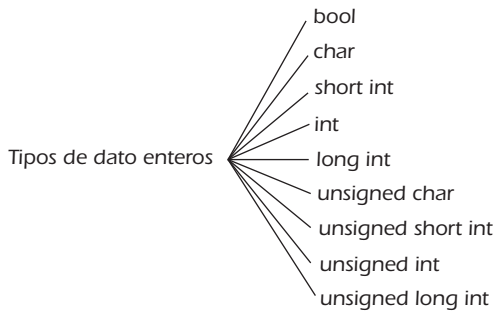


Figura 2.8 *Tipos de dato enteros en C++.*

El tipo de dato `int`

El conjunto de valores admitidos por el tipo de dato `int` son números enteros. Un valor entero consiste sólo de dígitos y de manera opcional puede estar precedido por un signo de más (+) o menos (–). Por tanto, un valor entero puede ser el número cero o cualquier valor numérico positivo o negativo sin un punto decimal. Ejemplos de enteros válidos son:

0 5 –10 +25 1000 253 –26 351 +36

Como ilustran estos ejemplos, los enteros pueden contener un signo explícito. No se permiten comas, puntos decimales ni símbolos especiales, cómo el signo de dólares. Son ejemplos de enteros inválidos:

$255.62 2 523 3. 6 243 892 1 492.89 +6.0

Los diferentes compiladores tienen su propio límite interno para los valores enteros más grande (más positivo) y más pequeño (más negativo) que pueden almacenarse en cada tipo de dato.[6] La asignación de almacenamiento más común es cuatro bytes para el tipo de dato `int`, lo cual restringe el conjunto de valores permitido en este tipo de dato a representar enteros en el rango de `-2,147,483,648` a `2,147,483,647`.[7]

El tipo de dato `char`

El tipo de dato `char` se usa para almacenar caracteres individuales. Los caracteres incluyen las letras del alfabeto (mayúsculas y minúsculas), los diez dígitos 0 a 9 y símbolos especiales como los siguientes: + $. , – y ! Un valor de carácter individual es cualquier letra, dígito o símbolo especial encerrado entre comillas sencillas. Los siguientes son ejemplos de valores de carácter válidos:

```
'A'   '$'   'b'   '7'   'y'   '!'   'M'   'q'
```

[6]Los límites impuestos por el compilador pueden encontrarse en el archivo de encabezado `limits` y se definen como las constantes hexadecimales `int_max` e `int_min`.

[7]En todos los casos, la magnitud del número entero más negativo siempre es un número mayor que la magnitud del entero más positivo. Esto se debe al método de almacenamiento de enteros de complemento a dos, el cual se describió en la sección 1.6.

Punto de Información

Datos atómicos
Un **valor de dato atómico** es un valor considerado una entidad completa en sí misma y que no puede descomponerse en un tipo de dato más pequeño. Por ejemplo, aunque un entero puede descomponerse en dígitos individuales, C++ no posee un tipo de dígito numérico. Más bien, cada entero es considerado como un valor completo por sí mismo y, como tal, se considera un dato atómico. Del mismo modo, debido a que el tipo de dato entero sólo admite valores de datos atómicos, se dice que es un **tipo de dato atómico**. Como se podría esperar, todos los tipos de datos integrados son tipos de datos atómicos.

Los valores de carácter se almacenan por lo general en una computadora usando los códigos ASCII o Unicode. ASCII (se pronuncia as-ki) son las siglas de código estándar americano para intercambio de informacion (American Standard Code for Information Interchange). El código ASCII proporciona códigos para un conjunto de caracteres basado en el idioma inglés, más códigos para control de impresión y de despliegue, como códigos de nueva línea y de expulsión del papel de la impresora. Cada código de carácter está contenido dentro de un solo byte, lo cual proporciona 256 códigos distintos. La tabla 2.3 enumera los códigos en byte ASCII para las letras mayúsculas.

Además, C++ proporciona el código Unicode más nuevo que usa dos bytes por carácter y puede representar 65 536 caracteres. Este código se usa para aplicaciones internacionales al proporcionar conjuntos de caracteres en otros idiomas además del inglés. Como los primeros 256 códigos de Unicode tienen el mismo valor numérico que los 256 códigos ASCII (el byte adicional se codifica con ceros), no es necesario preocuparse por cuál código de almacenamiento se usa cuando se utilizan caracteres en inglés.

Tabla 2.3 Los códigos ASCII para las letras mayúsculas

Letra	Código ASCII	Letra	Código ASCII
A	01000001	N	01001110
B	01000010	O	01001111
C	01000011	P	01010000
D	01000100	Q	01010001
E	01000101	R	01010010
F	01000110	S	01010011
G	01000111	T	01010100
H	01001000	U	01010101
I	01001001	V	01010110
J	01001010	W	01010111
K	01001011	X	01011000
L	01001100	Y	01011001
M	01001101	Z	01011010

Usando la tabla 2.3, podemos determinar cómo se almacenan dentro de una computadora los caracteres ‘B’, ‘A’, ‘R’, ‘T’, ‘E’ y ‘R’, por ejemplo, usando el código de caracteres ASCII. Esta secuencia de seis caracteres requiere seis bytes de almacenamiento (un byte por cada letra) y se almacenaría como se ilustra en la figura 2.9.

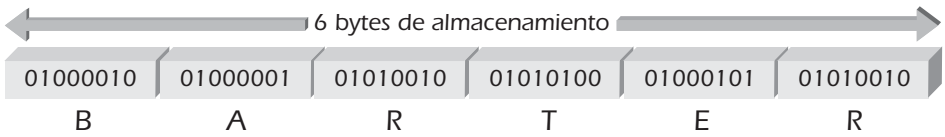


Figura 2.9 *Las letras BARTER almacenadas dentro de una computadora.*

El carácter escape

Un carácter que tiene un significado especial en C++ es la diagonal inversa, \, que se conoce como **carácter de escape**. Cuando se coloca este carácter directamente frente a un grupo selecto de caracteres, esto indica al compilador que escape de la forma en que estos caracteres se interpretarían en forma normal. La combinación de una diagonal inversa y estos caracteres específicos se llama **secuencia de escape**. Se ha mostrado un ejemplo de ésta en la secuencia de escape de línea nueva, `'\n'`, en el capítulo 1. La tabla 2.4 enumera las secuencias de escape más comunes en C++.

Tabla 2.4 Secuencias de escape

Secuencia de escape	Carácter representado	Significado	Código ASCII
`\n`	Línea nueva	Se mueve a una línea nueva	`00001010`
`\t`	Tabulador horizontal	Se mueve a la siguiente posición del tabulador horizontal	`00001001`
`\v`	Tabulador vertical	Se mueve a la siguiente posición del tabulador vertical	`00001011`
`\b`	Retroceso	Retrocede un espacio	`00001000`
`\r`	Retorno de carro	Mueve el cursor al inicio de la línea actual; se escribe para sobrescribir	`00001101`
`\f`	Alimentación de forma	Expulsa una hoja para iniciar otra	`00001100`
`\a`	Alerta	Emite una alerta (por lo general un sonido de campana)	`00000111`
`\\`	Diagonal inversa	Inserta un carácter de diagonal inversa (ésta se usa para colocar un carácter de diagonal inversa real dentro de una cadena)	`01011100`
`\?`	Signo de interrogación	Inserta un carácter de signo de interrogación	`00111111`
`\'`	Comilla sencilla	Inserta un carácter de comilla sencilla (ésta se usa para colocar una comilla sencilla interior dentro de un conjunto de comillas sencillas exteriores)	`00100111`
`\"`	Comillas dobles	Inserta un carácter de comilla doble (ésta se usa para colocar una comilla doble interior dentro de un conjunto de comillas dobles exteriores)	`00100010`
`\`*`nnn`	Número octal	El número *`nnn`* (*`n`* es un dígito) se considerará un número octal	—
`\x`*`hhhh`	Número hexadecimal	El número *`hhhh`* (*`h`* es un dígito) se considerará un número hexadecimal	—
`\0`	Carácter nulo	Inserta el carácter Null, el cual se define con un valor de 0	`00000000`

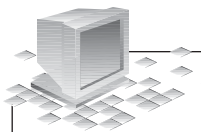
Aunque cada secuencia de escape enumerada en la tabla 2.4 está formada por dos caracteres distintos, la combinación de ambos, sin un espacio en blanco entre ellos, causa que el compilador cree el código individual enumerado en la columna de código ASCII de la tabla 2.4.

El tipo de datos `bool`

En C++, el tipo de datos `bool` se usa para representar datos booleanos (lógicos). Por ello, este tipo de datos está restringido a uno de dos valores: verdadero o falso. Este tipo de datos es más útil cuando un programa debe examinar una condición específica y, debido a la condición de ser verdadera o falsa, tomar un curso de acción prescrito. Por ejemplo, en una aplicación de ventas, la condición que se está examinando podría ser "la compra total es por $100 o más". Sólo cuando esta condición es verdadera se aplica un descuento. Sin embargo, debido a que un tipo de datos booleano usa un código de almacenamiento de enteros, tiene implicaciones útiles explotadas por casi todos los programadores profesionales en C++. Los usos prácticos de las condiciones booleanas se consideran en el capítulo 4, así que pospondremos la exposición de los datos booleanos hasta entonces.

Determinación del tamaño de almacenamiento

Una característica única de C++ es que le permite al programador ver dónde y cómo se almacenan los valores. Por ejemplo, C++ proporciona un operador denominado `sizeof()` que proporciona el número de bytes usados para almacenar valores para cualquier nombre de tipo de datos incluidos dentro de los paréntesis del operador. (Repase la sección 1.6 si no está familiarizado con el concepto de byte.) Éste es un operador integrado que no usa un símbolo aritmético para ejecutar su operación. El programa 2.5 usa este operador para determinar la cantidad de almacenamiento reservado para los tipos de datos, `int`, `char` y `bool`.



Programa 2.5

```
#include <iostream>
using namespace std;

int main()
{
  cout << "\nTipo de datos  Bytes"
       << "\n---------  -----"
       << "\nint          " << sizeof(int)
       << "\nchar         " << sizeof(char)
       << "\nbool         " << sizeof(bool)
       << '\n';

    return 0;
}
```

Punto de Información

El carácter `'\n'` y la cadena `"\n"`

El compilador reconoce tanto `'\n'` como `"\n"` como el carácter línea nueva. La diferencia está en el tipo de datos que se usa. De manera formal, `'\n'` es un carácter literal, mientras `"\n"` es una cadena literal. Desde un punto de vista práctico, ambos hacen que suceda lo mismo: Una línea nueva es forzada en el despliegue de salida. Sin embargo, al encontrar el valor de carácter `'\n'`, el compilador lo traduce usando el código de byte único 00001010 (véase la tabla 2.4). Al encontrar el valor de cadena `"\n"`, el compilador traduce esta cadena usando el código de carácter correcto pero también agrega un carácter extra de fin de cadena, el cual es `'\0'`.

La buena práctica de programación requiere que se termine el último despliegue de salida con una secuencia de escape de línea nueva. Esto asegura que la primera línea de salida de un programa no termina en la última línea desplegada por el programa ejecutado con anterioridad.

Al revisar el programa 2.5, notará que se inserta un solo valor de carácter en `cout` al encerrarlo dentro de comillas sencillas, como en la inserción de la secuencia de escape `'\n'` al final de la instrucción `cout`. Dentro de las primeras cinco líneas desplegadas, este carácter se incluye dentro de cada cadena de salida. Cada vez que el compilador encuentra la secuencia de escape de línea nueva, como un carácter individual o como parte de una cadena, la traduce como un solo carácter que fuerza que el despliegue empiece en una línea nueva. Aunque pueden usarse comillas dobles para la inserción de línea nueva final, como `"\n"`, esto designaría una cadena. Debido a que sólo se transmite un carácter, y para enfatizar que los caracteres únicos se designan usando comillas sencillas, se ha usado `'\n'` en lugar de `"\n"`. Desde un punto de vista práctico, sin embargo, ambas notaciones forzarán una línea nueva en la pantalla.

La salida del programa 2.5 depende del compilador. Es decir, cada compilador reportará en forma correcta la cantidad de almacenamiento que proporciona para el tipo de datos con el que está trabajando. Cuando se ejecutó en la computadora del autor, la cual usa el compilador actual Visual C++.net de Microsoft, se produjo la siguiente salida:

```
Tipos de datos   Bytes
--------------   -----
inf                4
char               1
bool               1
```

Para esta salida, la cual es el almacenamiento típico proporcionado por casi todos los compiladores C++ actuales, podemos determinar el rango de valores que pueden almacenarse en cada uno de estos tipos de datos `int`. Hacerlo, sin embargo, requiere comprender la diferencia entre un tipo de datos con signo y uno sin signo.

Punto de Información

Programas orientados a objetos y orientados a procedimientos

Con excepción del tipo booleano, todos los tipos de datos integrados de C++ son trasladados directo del lenguaje de procedimientos de C. No es sorprendente que los programas que sólo usen tipos integrados individuales no serán programas orientados a objetos. Más bien, como en el programa 2.5, se vuelven programas por procedimientos, es decir, un programa basado primordialmente en procedimientos, como `main()`.

Sólo cuando los tipos integrados se unen para formar un paquete de datos, el cual se convierte en un objeto, puede existir un programa orientado a objetos.

Tipos de datos con signo y sin signo

Un **tipo de datos con signo** se define como uno que permite almacenar valores negativos además del cero y valores positivos. Por ello, el tipo de datos `int` es un tipo de datos con signo. Un **tipo de datos sin signo** es uno que prevé sólo valores no negativos (es decir, cero y positivos).

Hay casos, sin embargo, donde una aplicación podría requerir sólo valores numéricos sin signo. Por ejemplo, muchas aplicaciones de fecha las almacenan en la forma numérica *añomesdía* (por tanto, la fecha 25/12/2007 se almacenaría como 20071225) y sólo procesan fechas después de 0. Para tales aplicaciones, las cuales nunca requerirán un valor negativo, puede usarse un tipo de datos sin signo.

Todos los tipos enteros sin signo, como `unsigned int`, proporcionan un rango de valores positivos que es, para todos los propósitos prácticos, el doble del rango provisto por su contraparte los enteros con signo. Este rango positivo extra queda disponible al usar el rango negativo de su versión con signo para números positivos adicionales.

Mediante la comprensión de la diferencia entre un tipo de datos con signo y sin signo, la tabla 2.5 puede utilizarse para determinar el rango de valores enteros admitido por los compiladores actuales de C++.

En la tabla 2.5, un `long int` usa la misma cantidad de almacenamiento (cuatro bytes) que un `int`. El único requerimiento del estándar ANSI de C++ es que un `int` debe proporcionar al menos tanto almacenamiento como un `short int`, y un `long int` debe proporcionar al menos tanto almacenamiento como un `int`. En los primeros sistemas de computadora de escritorio (década de los años 80), las cuales estaban limitadas en su capacidad de memoria a miles de bytes, un `short int` usaba de manera típica un byte de almacenamiento, un `int` dos bytes y un `long int` cuatro bytes. Este almacenamiento limitaba el rango de valores `int` de –32 768 a +32 767, mientras el uso de un `unsigned int` proporcionaba un rango de valores de 0 a 65 535, duplicando por tanto el número de valores positivos posibles, lo cual era considerable. Con el rango actual de valores `int` que abarca de –2 a +2 mil millones, la duplicación de valores positivos rara vez es algo que hay que tomar en cuenta. Además, usar un `long int` se vuelve innecesario porque usa la misma capacidad de almacenamiento que un `int`.

Tabla 2.5 Almacenamiento de tipo de datos enteros

Nombre del tipo de datos	Tamaño del **almacenamiento** (en bytes)	Rango de valores
`char`	1	256 caracteres
`bool`	1	verdadero (lo cual es considerado como cualquier valor positivo) y falso (lo cual es un cero)
`short int`	2	–32,768 a +32,767
`unsigned short int`	2	0 a 65,535
`int`	4	–2,147,483,648 a +2,147,483,647
`unsigned int`	4	0 a 4,294,967,295
`long int`	4	–2,147,483,648 a +2,147,483,647
`unsigned long int`	4	0 a 4,294,967,295

Tipos de punto flotante

Un ***número de punto flotante***, al cual se le llama ***número real***, puede ser el número cero o cualquier número positivo o negativo que contenga un punto decimal. Los siguientes son ejemplos de números de punto flotante:

```
+10.625    5.    -6.2    3251.92    0.0    0.33    -6.67    +2.
```

Los números 5., 0.0 y +2. se clasifican como valores de punto flotante, pero los mismos números escritos sin un punto decimal (5, 0, +2) serían valores enteros. Como con los valores enteros, en éstos no se permiten símbolos especiales como el signo de dólar y la coma en los números reales. Los siguientes son ejemplos de números reales inválidos:

```
5,326.25    24    6,459    $10.29    7.007.645
```

C++ acepta tres tipos de datos de punto flotante: `float`, `double` y `long double`. La diferencia entre estos tipos de datos es la cantidad de almacenamiento que usa un compilador para cada tipo. La mayor parte de los compiladores usan el doble de cantidad de almacenamiento para dobles que para flotantes, lo cual permite que un `double` tenga aproximadamente el doble de precisión de un `float`. Por esta razón, a veces se hace referencia a un valor `float` como un número de ***precisión simple*** y a un valor `double` como un número de ***precisión doble***. La asignación de almacenamiento real para cada tipo de datos, sin embargo, depende del compilador particular. El estándar ANSI C++ sólo requiere que un `double` tenga al menos la misma cantidad de precisión que un `float` y un `long double` tenga al menos la misma cantidad de almacenamiento que un `double`. En la actualidad, la mayor parte de los compiladores C++ asignan cuatro bytes para el tipo de datos `float` y ocho bytes para los tipos de datos `double` y `long double`. Esto produce el rango de números mostrado en la tabla 2.6.

Tabla 2.6 Tipos de datos de punto flotante

Tipo	Almacenamiento	Rango absoluto de valores (+ y –)
`float`	`4 bytes`	`1.40129846432481707e-45 a 3.40282346638528860e+38`
`double y long double`	`8 bytes`	`4.94065645841246544e-324 a 1.79769313486231570e+308`

En compiladores que usan la misma cantidad de almacenamiento para números double y long double, estos dos tipos de datos se vuelven idénticos. (El operador sizeof() que se usó en el programa 2.5 siempre puede usarse para determinar la cantidad de almacenamiento reservada por su compilador para estos tipos de datos.) Una literal tipo `float` se indica al añadir una f o F después del número y se crea un `long double` al añadir una l o L al número. En ausencia de estos sufijos, un número de punto flotante por defecto se establece como `double`. Por ejemplo, observe lo siguiente:

9.234 indica una doble literal
9.234F indica una literal flotante
9.234L indica una literal de doble largo

La única diferencia en estos números es la cantidad de almacenamiento que puede usar una computadora para almacenarlos. El apéndice C describe el formato de almacenamiento binario usado para números de punto flotante y su impacto en la precisión de los números.

Notación exponencial

Los números de punto flotante pueden escribirse en notación exponencial, la cual es semejante a la notación científica y se usa por lo común para expresar valores grandes y pequeños en forma compacta. Los siguientes ejemplos ilustran cómo pueden expresarse números con decimales en notación exponencial y científica:

Notación decimal	Notación exponencial	Notación científica
1625.	1.625e3	1.625×10^3
63421.	6.3421e4	6.3421×10^4
.00731	7.31e-3	7.31 x 10-3
.000625	6.25e-4	6.25 x 10-4

En notación exponencial, la letra e representa exponente. El número que sigue a la e representa una potencia de 10 e indica el número de lugares que debería moverse el punto decimal para obtener el valor decimal estándar. El punto decimal se mueve a la derecha si el número después de la e es positivo o se mueve a la izquierda si el número después de la e es negativo. Por ejemplo, el e3 en 1.625e3 significa mover el punto decimal tres lugares a la derecha, de modo que el número se vuelve 1625. El e–3 en 7.31e–3 significa mover el punto decimal tres lugares a la izquierda de modo que 7.31e–3 se convierte en .00731.

Punto de Información

¿Qué es precisión?

En teoría numérica, el término **precisión** por lo general se refiere a exactitud numérica. En este contexto, se usa una declaración como "este cálculo es exacto, o preciso, hasta el quinto lugar decimal". Esto significa que el quinto dígito después del punto decimal ha sido redondeado y el número es exacto dentro de ±0.00005.

En programación de computadoras, la precisión puede referirse a la exactitud de un número o a la cantidad de dígitos significativos en el número, donde los dígitos significativos se definen como el número de dígitos claramente correctos más 1. Por ejemplo, si el número 12.6874 se ha redondeado hasta el cuarto lugar decimal, es correcto decir que este número es preciso (es decir, exacto) hasta el cuarto lugar decimal. En otras palabras, todos los dígitos en el número son exactos excepto el cuarto dígito decimal, el cual ha sido redondeado. Del mismo modo, este mismo número tiene una precisión de seis dígitos, lo cual significa que los primeros cinco dígitos son correctos y el sexto dígito ha sido redondeado. Otra forma de decir esto es que el número 12.6874 tiene seis dígitos significativos.

Los dígitos significativos en un número no necesitan tener alguna relación con el número de dígitos desplegado. Por ejemplo, si el número 687.45678921 tiene cinco dígitos significativos, sólo es exacto hasta el valor 687.46, donde se supone que el último dígito está redondeado. De una manera similar, los valores en dólares en muchas aplicaciones financieras grandes con frecuencia son redondeados al centenar de miles de dólares más cercano. En tales aplicaciones, un valor en dólares desplegado de $12 400 000, por ejemplo, no es preciso hasta el dólar más cercano. Si este valor se especifica con tres dígitos significativos, sólo es exacto hasta el dígito de cientos de miles.

Ejercicios 2.3

1. Determine los tipos de datos apropiados para los siguientes datos:
   a. el promedio de cuatro calificaciones
   b. el número de días en un mes
   c. la longitud del puente Golden Gate
   d. los números en una lotería estatal
   e. la distancia de Brooklyn, N.Y., a Newark, N.J.
   f. el prefijo de carácter único que especifica un tipo de componente

2. Convierta los siguientes números en forma decimal estándar:

   6.34e5 1.95162e2 8.395e1 2.95e–3 4.623e–4

3. Escriba los siguientes números decimales usando notación exponencial:

   126. 656.23 3426.95 4893.2 .321 .0123 .006789

4. Compile y ejecute el programa 2.5 en su computadora.

5. Modifique el programa 2.5 para determinar el almacenamiento usado por su compilador para todos los tipos de datos enteros de C++.

6. Usando los manuales de referencia del sistema para su computadora, determine el código de caracteres usado por su computadora.

7. Muestre cómo se almacenaría el nombre KINGSLEY dentro de una computadora que usa el código ASCII. Es decir, trace un dibujo similar a la figura 2.9 para el nombre KINGSLEY.

8. Repita el ejercicio 7 usando las letras de su propio apellido.

9. Modifique el programa 2.5 para determinar cuántos bytes asigna su compilador a los tipos de datos `float`, `double` y `long double`.

10. Debido a que las computadoras usan diferentes representaciones para almacenar valores enteros, de punto flotante, de precisión doble y de carácter, analice cómo un programa podría alertar a la computadora de los tipos de datos de los diversos valores que usará.

11. Aunque nos hemos concentrado en operaciones que implican números enteros y de punto flotante, C++ permite que se sumen o resten caracteres y enteros. Esto puede hacerse porque un carácter se almacena usando un código de enteros (es un tipo de datos entero). Por tanto, los caracteres y enteros pueden mezclarse con libertad en expresiones aritméticas. Por ejemplo, si su computadora usa el código ASCII, la expresión 'a' + 1 es igual a 'b', y 'z' – 1 es igual a 'y'. Del mismo modo, 'A' + 1 es 'B' y 'Z' – 1 es 'Y'. Con esto como antecedente, determine el carácter que resulta de las siguientes expresiones. (Asuma que todos los caracteres se almacenan usando el código ASCII.)
    a. 'm' – 5
    b. 'm' + 5
    c. 'G' + 6
    d. 'G' – 6
    e. 'b' – 'a'
    f. 'g' – 'a' + 1
    g. 'G' – 'A' + 1

NOTA: Para completar el siguiente ejercicio, necesita comprender los conceptos básicos del almacenamiento en las computadoras. De manera específica, si no está familiarizado con el concepto de byte, refiérase a la sección 1.6 antes de realizar el siguiente ejercicio.

12. Aunque el número total de bytes varía de una computadora a otra, son comunes los tamaños de memoria de 65 536 a más de varios millones de bytes. En lenguaje de computadora, la letra K representa el número 1 024, el cual es 2 elevado a la décima potencia, y M representa el número 1 048 576, el cual es 2 elevado a la vigésima potencia. Por tanto, un tamaño de memoria de 640K en realidad es 640 por 1024, o 655 360 bytes, y un tamaño de memoria de 4M en realidad es 4 por 1 048 576, lo cual es 4 194 304 bytes. Usando esta información, calcule el número total de bytes en lo siguiente:
    a. una memoria que contiene 128M bytes
    b. una memoria que contiene 256M bytes
    c. una memoria que contiene 512M bytes
    d. una memoria que consta de 256M palabras, donde cada palabra es de 2 bytes
    e. una memoria que consta de 256M palabras, donde cada palabra es de 4 bytes
    f. un disco que especifica 1.44M bytes
    g. un disco que especifica 250MB

2.4 OPERACIONES ARITMÉTICAS

En la sección anterior se presentaron los valores de datos correspondientes para cada tipo de datos integrado de C++. En esta sección, se proporciona el conjunto de operaciones aritméticas que pueden aplicarse a estos valores.

Los números enteros y reales pueden sumarse, restarse, multiplicarse y dividirse. Aunque por lo general es mejor no mezclar números enteros y reales cuando se realizan operaciones aritméticas, se obtienen resultados predecibles cuando se usan diferentes tipos de datos en la misma expresión aritmética. Es sorprendente que se puedan sumar, o restar, datos de carácter a datos de carácter y enteros para producir resultados útiles. (Por ejemplo, 'A' + 1 produce el carácter 'B'.) Esto es posible debido a que los caracteres se almacenan usando códigos de almacenamiento de números enteros.

Los operadores usados en operaciones aritméticas se llaman **operadores aritméticos** y son los siguientes:

Operación	Operador
Adición	+
Sustracción	–
Multiplicación	*
División	/
División de módulo	%

No hay que preocuparse en esta etapa si no entiende el término "división de módulo". Aprenderá más sobre este operador más adelante en esta sección.

Estos operadores se conocen como **operadores binarios**. Este término refleja el hecho que el operador requiere dos operandos para producir un resultado. Un **operando** puede ser un valor literal o un identificador que tiene un valor asociado con él. Una **expresión aritmética binaria simple** consta de un operador aritmético binario que conecta dos valores literales en la forma:

```
valorLiteral operador valorLiteral
```

Los siguientes son ejemplos de expresiones aritméticas binarias simples:

```
3 + 7
18 - 3
12.62 + 9.8
.08 * 12.2
12.6 / 2.
```

Los espacios alrededor de los operadores aritméticos en estos ejemplos se insertan estrictamente para ganar en claridad y pueden omitirse sin afectar el valor de la expresión. Una expresión en C++ debe ser introducida en forma de una línea recta. Por tanto, por ejemplo, la expresión de C++ equivalente a 12.6 dividido entre 2 debe introducirse como 12.6 / 2 y no como la expresión algebraica

$$\frac{12.6}{2}$$

Puede usar `cout` para desplegar el valor de cualquier expresión aritmética en la pantalla de la computadora. Para hacer esto, el valor deseado debe transladarse al objeto. Por ejemplo, esta instrucción produce el despliegue 21:

```
cout << (6 + 15);
```

En sentido estricto, los paréntesis que rodean a la expresión 6 + 15 no se requieren para indicar que el valor de la expresión (es decir, 21) se está desplegando.[8] Además de desplegar un valor numérico, `cout` puede desplegar una cadena identificando la salida, como se hizo en la sección 1.3. Por ejemplo, esta instrucción causa que dos piezas de datos, una cadena y un valor, se envíen a `cout`:

```
cout << "La suma de 6 y 15 es " <<  (6 + 15);
```

De manera individual, cada conjunto de datos enviados a `cout` debe ir precedido por su propio símbolo de inserción de operador (<<). Aquí, los primeros datos enviados para despliegue es la cadena `"La suma de 6 y 15 es "`, y el segundo elemento enviado es el valor de la expresión 6 + 15. El despliegue producido por esta instrucción es el siguiente:

```
La suma de 6 y 15 es 21
```

El espacio entre la palabra "es" y el número 21 se da por el espacio colocado dentro de la cadena pasada a `cout`. En lo que respecta a `cout`, su entrada es un conjunto de caracteres que son enviados para ser desplegados en el orden en que son recibidos. Los caracteres de la entrada hacen cola, uno detrás de otro, y son enviados a la consola para su despliegue. Colocar un espacio en la entrada provoca que este espacio sea parte del flujo de caracteres que se despliega. Por ejemplo, examine la siguiente instrucción:

```
cout << "La suma de 12.2 y 15.754 es " <<  (12.2 + 15.754);
```

Que produce el siguiente despliegue:

```
La suma de 12.2 y 15.754 es 27.954
```

Cuando se hacen múltiples inserciones a `cout`, el código puede extenderse a lo largo de múltiples líneas. Sin embargo, sólo debe usarse un punto y coma, el cual se coloca después de la última inserción y termina la instrucción completa. Por tanto, el despliegue anterior es producido por la siguiente instrucción:

```
cout << "La suma de 12.2 y 15.754 es "
     <<  (12.2 + 15.754);
```

Sin embargo, cuando permite que una instrucción así ocupe múltiples líneas, deben seguirse dos reglas: una cadena contenida entre comillas no puede extenderse más de una línea, y el punto y coma final sólo deberá aparecer en la última línea. Siempre pueden colocarse múltiples símbolos de inserción dentro de una línea.

Los números de punto flotante se despliegan con suficientes lugares decimales a la derecha del punto decimal para acomodar la parte fraccionaria del número. Esto es cierto si el número tiene seis o menos dígitos decimales. Si el número tiene más de seis dígitos decima-

[8]Esto se debe a que el operador + tiene una precedencia mayor que el operador <<; por tanto, la adición se ejecuta antes de la inserción.

les, la parte fraccionaria se redondea a seis dígitos decimales, y si el número no tiene dígitos decimales, no se desplegarán ni un punto decimal ni algún dígito decimal.[9]

El programa 2.6 ilustra el uso de `cout` para desplegar los resultados de expresiones aritméticas dentro de las instrucciones de un programa completo.

Programa 2.6

```
#include <iostream>
using namespace std;

int main()
{
  cout << "15.0 más 2.0 es igual a "            << (15.0 + 2.0) << endl
       << "15.0 menos 2.0 es igual a "          << (15.0 - 2.0) << endl
       << "15.0 por 2.0 es igual a "            << (15.0 * 2.0) << endl
       << "15.0 dividido entre 2.0 es igual a " << (15.0 / 2.0) << endl;

  return 0;
}
```

La salida del programa 2.6 es la siguiente:

```
15.0 más 2.0 es igual a 17
15.0 menos 2.0 es igual a 13
15.0 por 2.0 es igual a 30
15.0 dividido entre 2.0 es igual a 7.5
```

El único elemento nuevo presentado en el programa 2.6 es el término `endl`, el cual es un ejemplo de un manipulador C++. Un ***manipulador*** es un elemento usado para manipular cómo se despliega el flujo de salida de caracteres. En particular, el manipulador `endl` provoca que primero se inserte un carácter de línea nueva (`'\n'`) en el despliegue y obliga a que todas las inserciones actuales se desplieguen de inmediato, en lugar de esperar por más datos. (La sección 3.2 contiene una lista de los manipuladores más comunes.)

Tipos de expresión

Una ***expresión*** es cualquier combinación de operadores y operandos que pueden ser evaluados para producir un valor. Una expresión que contiene sólo valores enteros como operandos se llama ***expresión entera***, y el resultado de la expresión es un valor entero. Del mismo modo, una expresión que sólo contiene valores de punto flotante (de precisión simple y precisión

[9]No se define ninguna de estas salidas como parte del lenguaje C++. Más bien es definida por un sistema de clases y de rutinas proporcionadas de cada recopilador de C++.

doble) como operandos se llama **expresión de punto flotante**, y el resultado de una expresión así es un valor de punto flotante (también se usa el término **expresión real**). Una expresión que contiene valores enteros y de punto flotante se llama **expresión en modo mixto**. Aunque por lo general es mejor no mezclar valores enteros y de punto flotante en una operación aritmética, el tipo de datos de cada operación se determina por las siguientes reglas:

1. Si ambos operandos son enteros, el resultado de la operación es un entero.
2. Si un operando es un valor real, el resultado de la operación es un valor de precisión doble.

El resultado de una expresión aritmética nunca es un número de precisión simple (`float`). Esto se debe a que, durante la ejecución, un programa C++ convierte de manera temporal todos los números de precisión simple a números de precisión doble cuando se está evaluando una expresión aritmética.

División de enteros

La división de dos valores enteros puede producir resultados bastante extraños para los incautos. Por ejemplo, la expresión 15/2 produce el resultado entero 7. Debido a que los enteros no pueden contener una parte fraccionaria, no puede obtenerse un valor de 7.5. La parte fraccionaria obtenida cuando se dividen dos enteros, es decir, el residuo, siempre se elimina (se trunca). Por tanto, el valor de 9/4 es 2, y 20/3 es 6.

Sin embargo, con frecuencia puede ser necesario conservar el residuo de una división de enteros. Para hacer esto, C++ proporciona un operador aritmético que tiene el símbolo %. Este operador, llamado **módulo** (y también conocido como **operador de residuo**), captura el residuo cuando un número entero es dividido entre un entero (usar un valor no entero con el operador de módulo produce un error en el compilador):

```
 9 % 4 es 1  (es decir, el residuo cuando 9 se divide entre 4 es 1)
17 % 3 es 2  (es decir, el residuo cuando 17 se divide entre 3 es 2)
15 % 4 es 3  (es decir, el residuo cuando 15 se divide entre 4 es 3)
14 % 2 es 0  (es decir, el residuo cuando 14 se divide entre 2 es 0)
```

Con más precisión, el operador de módulo determina primero el número entero de veces que el dividendo, el cual es el número que sigue al operador %, puede dividirse entre el divisor, el cual es el número anterior al operador %. Entonces devuelve el residuo.

Negación

Además de los operadores aritméticos binarios, C++ proporciona operadores unitarios. Un **operador unitario** opera sobre un operando individual. Uno de estos operadores unitario usa el mismo símbolo que la sustracción binaria (–). El signo de menos frente a un valor numérico individual niega el número (invierte su signo).

La tabla 2.7 resume las seis operaciones aritméticas que se han descrito hasta ahora y enumera el tipo de datos para el resultado producido por cada operador, basado en el tipo de datos de los operandos involucrados.

Punto de Información

El manipulador `endl`
En muchos sistemas, el manipulador `endl` y la secuencia de escape `\n` se procesan en la misma forma y producen el mismo efecto. La única excepción se da en aquellos sistemas donde la salida se acumula en forma interna hasta que se colectan suficientes caracteres para que sea ventajoso desplegarlos a todos en una ráfaga en la pantalla. En tales sistemas, los cuales se conocen como sistemas con memoria intermedia o "buffer", el manipulador `endl` fuerza a la salida acumulada a ser desplegada de inmediato, sin esperar que ningún carácter adicional llene el área de buffer antes de imprimirse. Prácticamente, no notará una diferencia en el despliegue final. Por tanto, como regla general, deberá usar la secuencia de escape `\n` siempre que pueda ser incluida dentro de una cadena existente y usar el manipulador `endl` siempre que aparezca `\n` por sí mismo o para determinar de manera formal el final de un grupo específico de despliegue de salida.

Tabla 2.7 Resumen de operadores aritméticos

Operación	Operador	Tipo	Operando	Resultado
Adición	+	Binario	Ambos son enteros	Entero
			Un operando no es un entero	Precisión doble
Sustracción	-	Binario	Ambos son enteros	Entero
			Un operando no es un entero	Precisión doble
Multiplicación	*	Binario	Ambos son enteros	Entero
			Un operando no es un entero	Precisión doble
División	/	Binario	Ambos son enteros	Entero
			Un operando no es un entero	Precisión doble
Módulo	%	Binario	Ambos son enteros	Entero
			Un operando no es un entero	Precisión doble
Negación	-	Unitario	Entero o doble	Igual que el operando

Precedencia del operador y asociatividad

Además de expresiones simples como 5 + 12 y .08 * 26.2, pueden crearse expresiones aritméticas más complejas. C++, como la mayor parte de otros lenguajes de programación, re-

quiere que se sigan ciertas reglas cuando se escriben expresiones que contengan más de un operador aritmético. Estas reglas son las siguientes:

1. Nunca deben colocarse dos símbolos de operadores aritméticos binarios uno al lado del otro. Por ejemplo, 5 * % 6 es inválido porque los dos operadores, * y %, están colocados uno junto al otro.
2. Pueden usarse paréntesis para formar agrupamientos, y todas las expresiones encerradas dentro de paréntesis son evaluadas primero. Esto permite a los paréntesis alterar la evaluación en cualquier orden deseado. Por ejemplo, en la expresión (6 + 4) / (2 + 3), 6 + 4 y 2 + 3 se evalúan primero para producir 10 / 5. Luego se evalúa 10 / 5 para producir 2.
3. Conjuntos de paréntesis pueden ser encerrados por otros paréntesis. Por ejemplo, la expresión (2 * (3 + 7)) / 5 es válida y evalúa para 4. Cuando se incluyen paréntesis dentro de paréntesis, las expresiones en los paréntesis interiores siempre se evalúan primero. La evaluación continúa desde los paréntesis más interiores hasta los más exteriores hasta que se han evaluado las expresiones en todos los paréntesis. El número de paréntesis de cierre,), siempre debe ser igual al número de paréntesis de apertura, (, de modo que no existan conjuntos sin par.
4. No pueden usarse paréntesis para indicar multiplicación; en su lugar, debe usarse el operador de multiplicación, *. Por ejemplo, la expresión (3 + 4) (5 + 1) es inválida. La expresión correcta es (3 + 4) * (5 + 1).

Los paréntesis deberían especificar agrupamientos lógicos de operandos e indicarle con claridad, al compilador y a los programadores, el orden previsto de las operaciones aritméticas. Aunque las expresiones dentro de paréntesis siempre se evalúan primero, las expresiones que contienen múltiples operadores, con y sin paréntesis, se evalúan por la prioridad, o **precedencia**, de los operadores. Hay tres niveles de precedencia:

- P1: todas las negaciones se realizan primero.
- P2: a continuación se calculan las operaciones de multiplicación, división y módulo. Las expresiones que contienen más de un operador de multiplicación, división o módulo se evalúan de izquierda a derecha conforme se encuentra cada operador. Por ejemplo, en la expresión 35 / 7 % 3 * 4, todas las operaciones son de la misma prioridad, así que las operaciones se ejecutarán de izquierda a derecha conforme se encuentre cada operador. Por tanto, la división se hace primero, produciendo la expresión 5 % 3 * 4. La operación de módulo se ejecuta a continuación, produciendo un resultado de 2. Por último, se calcula el valor de 2 * 4 para producir 8.
- P3: la adición y la sustracción se calculan al último. Las expresiones que contienen más de una adición o sustracción se evalúan de izquierda a derecha conforme se encuentre cada operador.

Además de la precedencia, los operadores tienen una ***asociatividad***, la cual es el orden en que se evalúan los operadores de la misma precedencia, como se describió en la regla P2. Por ejemplo, ¿la expresión 6.0 * 6/4 produce 9.0, lo cual es (6.0 * 6)/4: o 6.0, lo cual es 6.0 * (6/4)? La respuesta es 9.0, porque los operadores de C++ usan la misma asociatividad que en

las matemáticas generales, las cuales evalúan la multiplicación de izquierda a derecha, como lo indica la regla P2. La tabla 2.8 enumera la precedencia y asociatividad de los operadores considerados en esta sección. Como se ha visto, la precedencia de un operador establece su prioridad en relación con todos los demás operadores. Los operadores en la parte superior de la tabla 2.8 tienen una prioridad mayor que los operadores en la parte inferior de la tabla. En expresiones que contienen múltiples operadores de diferente precedencia, el operador con la mayor precedencia se usa antes que un operador con menor precedencia. Por ejemplo, en la expresión 6 + 4 / 2 + 3, debido a que el operador de división tiene una precedencia mayor (P2) que la adición, la división se hace primero, produciendo un resultado intermedio de 6 + 2 + 3. Entonces se ejecutan las adiciones, de izquierda a derecha, para producir un resultado final de 11.

Tabla 2.8 Precedencia y asociatividad de operadores

Operador	Asociatividad
unitario –	derecha a izquierda
* / %	izquierda a derecha
+ –	izquierda a derecha

Por último, usaremos la tabla 2.8 o las reglas de precedencia para evaluar una expresión que contiene operadores de diferente precedencia, como 8 + 5 * 7 % 2 * 4. Debido a que los operadores de multiplicación y módulo tienen una mayor precedencia que el operador de adición, estas dos operaciones se evalúan primero (P2), usando su asociatividad de izquierda a derecha, antes que se evalúe la adición (P3). Por tanto, la expresión completa se evalúa como sigue:

```
8 + 5 * 7 % 2 * 4 =
   8 + 35 % 2 * 4 =
        8 + 1 * 4 =
            8 + 4 = 12
```

Ejercicios 2.4

1. A continuación se enumeran expresiones algebraicas correctas y expresiones C++ incorrectas correspondientes a ellas. Encuentre los errores y escriba las expresiones C++ corregidas.

	Álgebra	*Expresión C++*
a.	`(2)(3) + (4)(5)`	`(2)(3) + (4)(5)`

b.	`6 + 18`
	`------`
	`  2`
		`6 + 18 / 2`

c.	`  4.5`
	`----------`
	`12.2 - 3.1`
		`4.5 / 12.2 - 3.1`

d. `4.6(3.0 + 14.9)` `4.6(3.0 + 14.9)`

e. `(12.1 + 18.9)(15.3 - 3.8)` `(12.1 + 18.9)(15.3 - 3.8)`

2. Determine el valor de las siguientes expresiones enteras:

a. 3 + 4 * 6
b. 3 * 4 / 6 + 6
c. 2 * 3 / 12 * 8 / 4
d. 10 * (1 + 7 * 3)
e. 20 – 2 / 6 + 3
f. 20 – 2 / (6 + 3)
g. (20 – 2) / 6 + 3
h. (20 – 2) / (6 + 3)
i. 50 % 20
j. (10 + 3) % 4

3. Determine el valor de las siguientes expresiones de punto flotante:

a. 3.0 + 4.0 * 6.0
b. 3.0 * 4.0 / 6.0 + 6.0
c. 2.0 * 3.0 / 12.0 * 8.0 / 4.0
d. 10.0 * (1.0 + 7.0 * 3.0)
e. 20.0 – 2.0 / 6.0 + 3.0
f. 20.0 – 2.0 / (6.0 + 3.0)
g. (20.0 – 2.0) / 6.0 + 3.0
h. (20.0 – 2.0) / (6.0 + 3.0)

4. Evalúe las siguientes expresiones en modo mixto y enumere el tipo de datos del resultado. Al evaluar las expresiones, esté consciente de los tipos de datos de todos los cálculos intermedios.

a. 10.0 + 15 / 2 + 4.3
b. 10.0 + 15.0 / 2 + 4.3
c. 3.0 * 4 / 6 + 6
d. 3 * 4.0 / 6 + 6
e. 20.0 – 2 / 6 + 3
f. 10 + 17 * 3 + 4
g. 10 + 17 / 3. + 4
h. 3.0 * 4 % 6 + 6
i. 10 + 17 % 3 + 4.

5. Suponga que `amount` almacena el valor entero 1, `m` almacena el valor entero 50, `n` almacena el valor entero 10 y `p` almacena el valor entero 5. Evalúe las siguientes expresiones:

a. `n / p + 3`
b. `m / p + n - 10 * amount`
c. `m - 3 * n + 4 * amount`
d. `amount / 5`
e. `18 / p`
f. `-p * n`
g. `-m / 20`
h. `(m + n) / (p + amount)`
i. `m + n / p + amount`

6. Repita el ejercicio 5, suponiendo que `amount` almacena el valor 1.0, `m` almacena el valor 50.0, `n` almacena el valor 10.0 y `p` almacena el valor 5.0.
7. Introduzca, compile y ejecute el programa 2.2 en su sistema de cómputo.
8. Determine la salida del siguiente programa:

```
#include <iostream>
using namespace std;

int main()  // un programa que ilustra el truncamiento de
               enteros
{
  cout << "respuesta1 es el entero " << 9/4;
  cout << "\nrespuesta2 es el entero " << 17/3;

  return 0;
}
```

9. Determine la salida del siguiente programa:

```
#include <iostream>
using namespace std;

int main()  // un programa que ilustra el operador %
{
  cout << "El residuo de 9 dividido entre 4 es " << 9 % 4;
  cout << "\nEl residuo de 17 dividido entre 3 es " << 17 % 3;

  return 0;
}
```

10. Escriba un programa C++ que despliegue los resultados de las expresiones 3.0 * 5.0, 7.1 * 8.3 - 2.2 y 3.2 / (6.1 * 5). Calcule el valor de estas expresiones en forma manual para verificar que los valores desplegados son correctos.
11. Escriba un programa C++ que despliegue los resultados de las expresiones 15 / 4, 15 % 4 y 5 * 3 - (6 * 4). Calcule el valor de estas expresiones en forma manual para verificar que los valores desplegados son correctos.

2.5 VARIABLES E INSTRUCCIONES DE DECLARACIÓN

Todos los valores enteros, de punto flotante y otros usados en un programa de computadora se almacenan en la unidad de memoria de la computadora y se recuperan de ella. Desde el punto de vista conceptual, las ubicaciones individuales de memoria en la unidad de memoria están ordenadas como las habitaciones en un gran hotel. Como en un hotel, cada ubicación de memoria tiene una dirección única ("número de habitación"). Antes que existieran lenguajes de alto nivel como C++, se hacía referencia a las ubicaciones de memoria por sus direcciones. Por ejemplo, almacenar los valores enteros 45 y 12 en las ubicaciones de memoria 1652 y 2548 (véase la figura 2.10), respectivamente, requería instrucciones equivalentes a

Coloque un 45 en la ubicación 1652
Coloque un 12 en la ubicación 2548

Para sumar los dos números que se acaban de almacenar y guardar el resultado en otra ubicación de memoria, por ejemplo en la ubicación 3000, se necesita una instrucción comparable a

Sume el contenido de la ubicación 1652
al contenido de la ubicación 2548
y almacene el resultado en la ubicación 3000

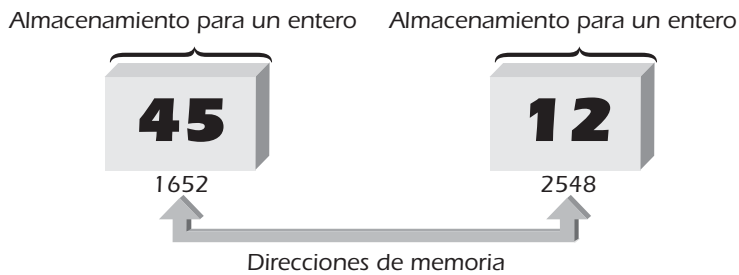


Figura 2.10 *Almacenamiento suficiente para dos enteros.*

Es evidente que este método de almacenamiento y recuperación es un proceso engorroso. En lenguajes de alto nivel como C++, se usan nombres simbólicos en lugar de direcciones de memoria reales. Estos nombres simbólicos se llaman **variables**. Una variable es tan sólo un nombre dado por el programador para referirse a ubicaciones de almacenamiento de la computadora. Se usa el término variable porque el valor almacenado en la variable puede cambiar, o variar. Para cada nombre que usa el programador, la computadora se mantiene al tanto de la dirección de memoria real correspondiente a ese nombre. En nuestra analogía de la habitación de hotel, esto es equivalente a poner un nombre en la puerta de una habitación y referirse a la habitación con ese nombre, como la habitación AZUL, en lugar de usar el número real de la habitación.

En C++, la selección de nombres de variables se deja al programador siempre que observe las reglas para seleccionar nombres de identificadores. Éstos se presentaron en la página 35 y se resumen a continuación.

1. El nombre de la variable debe comenzar con una letra o subrayado (_) y sólo puede contener letras, subrayados o dígitos. No puede contener ningún espacio en blanco, comas ni símbolos especiales, como ( ) & , $ # . ! \ ?
2. Un nombre de variable no puede ser una palabra clave (véase la tabla 2.1).
3. El nombre de la variable no puede consistir en más de 1024 caracteres.

Además, los nombres de variables deben ser mnemónicos que den algún indicio del uso de la variable. Por ejemplo, un buen nombre para una variable usada para almacenar un valor que es el total de algunos otros valores sería `suma` o `total`. No deberán seleccionarse nombres de variables que no dan ningún indicio del valor almacenado, como `r2d2`, `linda`, `beto` y `getum`. Como con los nombres de función, los nombres de variables pueden escribirse con letras mayúsculas y minúsculas.

Ahora suponga que a la primera ubicación de memoria ilustrada en la figura 2.11, la cual tiene la dirección 1652, se le da el nombre `num1`. También suponga que a la ubicación de memoria 2548 se le da el nombre de variable `num2`, y a la ubicación de memoria 3000 se le da el nombre de variable `total`, como se ilustra en la figura 2.11.

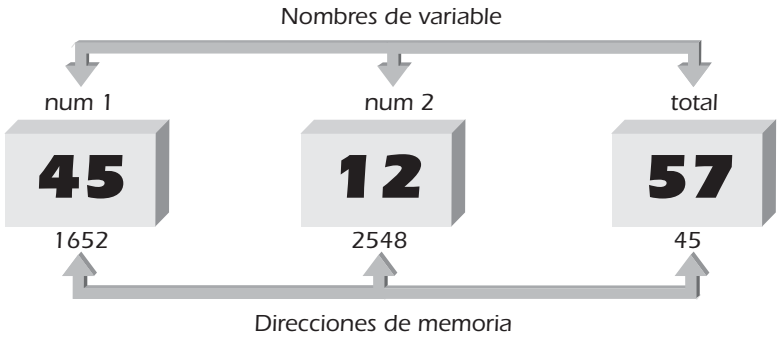


Figura 2.11 *Denominación de las ubicaciones de almacenamiento.*

Usando estos nombres de variable, la operación de almacenar 45 en la ubicación 1652, almacenar 12 en la ubicación 2548 y sumar los contenidos de estas dos ubicaciones se logra con las instrucciones de C++

```
num1 = 45;
num2 = 12;
total = num1 + num2;
```

Cada una de estas tres instrucciones se llama ***instrucción de asignación*** porque le indica a la computadora que asigne (almacene) un valor en una variable. Las instrucciones de asignación siempre tienen un signo de igual (=) y un nombre de variable inmediatamente a la izquierda de este signo. El valor a la derecha del signo de igual se determina primero, y este valor se asigna a la variable a la izquierda del signo de igual. Los espacios en blanco en las instrucciones de asignación se insertan para una mayor legibilidad. En el siguiente capítulo se hablará más sobre las instrucciones de asignación, pero por ahora pueden usarse para almacenar valores en variables.

Un nombre de variable es de utilidad porque libera al programador en lo que concierne a dónde se almacenan los datos en forma física dentro de la computadora. Tan sólo se usa el nombre de variable y se deja que el compilador se preocupe por el lugar de la memoria en que se almacenan los datos. Sin embargo, antes de almacenar un valor en una variable, C++ requiere que se declare con claridad el tipo de datos que se van a almacenar en ella. Debe indicarse al compilador, con anticipación, los nombres de las variables que se usarán para caracteres, los nombres que se usarán para enteros y los nombres que se usarán para almacenar los otros tipos de datos de C++.

Instrucciones de declaración

Nombrar una variable y especificar el tipo de datos que pueden almacenarse en ella se logra usando ***instrucciones de declaración***. Una instrucción de declaración tiene la forma general

tipo-de-datos nombreDeVariable;

Punto de Información

Datos atómicos
Las variables que declaramos aquí se han usado para almacenar valores de datos atómicos. Un valor de datos atómicos es aquel que se considera una entidad completa por sí misma y que no puede descomponerse en un tipo de datos más pequeño respaldado por el lenguaje. Por ejemplo, aunque un entero puede descomponerse en dígitos individuales, C++ posee un tipo de dígitos numéricos. Más bien, cada entero es considerado como un valor completo por sí mismo y, como tal, se considera dato atómico. Del mismo modo, en vista que el tipo de datos enteros sólo respalda valores de datos atómicos, se dice que es un tipo de datos atómicos. Como podría esperarse, double, char y bool también son tipos de datos atómicos.

donde *tipo-de-datos* designa un tipo de datos válidos en C++ y *nombreDeVariable* es un nombre de variable seleccionado por el usuario. Por ejemplo, las variables usadas para contener valores enteros se declaran usando la palabra clave `int` para especificar el tipo de datos y tiene la forma:

```
int nombreDeVariable;
```

Por tanto, la instrucción de declaración

```
int sum;
```

declara `suma` como el nombre de una variable capaz de almacenar un valor entero.

Además de la palabra reservada `int` usada para especificar un entero, la palabra reservada **`long`** se usa para especificar un entero largo.[10] Por ejemplo, la instrucción

```
long fechanum;
```

declara `fechanum` como una variable que se usará para almacenar un entero largo. Cuando se usa el calificador `long` puede incluirse la palabra clave `int`. Por tanto, la declaración anterior también puede escribirse como

```
long int fechanum;
```

Las variables usadas para contener valores de precisión simple se declaran usando la palabra clave `float`, mientras las variables que se usarán para contener valores de precisión doble se declaran usando la palabra clave `double`. Por ejemplo, la instrucción

```
float primernum;
```

declara `primernum` como una variable que se usará para almacenar un número de precisión simple. Del mismo modo, la instrucción

```
double segundonum;
```

declara que la variable `segundonum` se usará para almacenar un número de precisión doble.

Aunque las instrucciones de declaración pueden colocarse en cualquier parte dentro de una función, por lo general las declaraciones se agrupan y se colocan inmediatamente después de la llave de apertura de la función. En todos los casos, sin embargo, una variable de-

[10]Además, las palabras reservadas `unsigned int` se usan para especificar un entero que sólo puede almacenar números no negativos y la palabra reservada `short` se usa para especificar un entero corto.

be declararse antes que pueda usarse y, como todas las instrucciones de C++, las instrucciones de declaración deben terminar con un punto y coma. Si las instrucciones de declaración se colocan después de la llave de apertura de la función, una función `main()` simple que contenga instrucciones de declaración tendría la forma general

```
#include <iostream>
using namespace std;

int main()
{
  instrucciones de declaración;

  otras instrucciones;

  return 0;
}
```

El programa 2.7 ilustra esta forma al declarar y usar cuatro variables de precisión doble, usando el objeto `cout` para desplegar el contenido de una de las variables.

Programa 2.7

```
#include <iostream>
using namespace std;

int main()
{
  double calif1; // declara calif1 como una variable de precisión doble
  double calif2; // declara calif2 como una variable de precisión doble
  double total;   // declara total como una variable de precisión doble
  double promedio; // declara promedio como una variable de precisión doble

  calif1 = 85.5;
  calif2 = 97.0;
  total = calif1 + calif2;
  promedio = total/2.0;  // divide el total entre 2.0
  cout << "El promedio de las calificaciones es " << promedio << endl;

  return 0;
}
```

La colocación de las instrucciones de declaración en el programa 2.7 es simple, aunque pronto se verá que las cuatro declaraciones individuales pueden combinarse en una sola declaración. Cuando se ejecuta el programa 2.7, se despliega la siguiente salida:

```
El promedio de las calificaciones es 91.25
```

Hay que observar que cuando se inserta un nombre de variable en un objeto `cout`, el valor almacenado en la variable se coloca en el flujo de salida y se despliega.

Del mismo modo en que las variables de números enteros y reales (de precisión simple, de precisión doble y de doble largo) deben declararse antes que puedan utilizarse, también debe declararse una variable usada para almacenar un carácter individual. Las variables de carácter se declaran usando la palabra reservada `char`. Por ejemplo, la declaración

```
char ch;
```

declara que `ch` es una variable de carácter. El programa 2.8 ilustra esta declaración y el uso de `cout` para desplegar el valor almacenado en una variable de carácter.

Programa 2.8

```
#include <iostream>
using namespace std;

int main()
{
  char ch;     // esto declara una variable de carácter

  ch = 'a';    // almacena la letra a en ch
  cout << "El carácter almacenado en ch es " << ch << endl;
  ch = 'm';    // ahora almacena la letra m en ch
  cout << "El carácter almacenado ahora en ch es "<< ch << endl;

  return 0;
}
```

Cuando se ejecuta el programa 2.8, la salida producida es:

```
El carácter almacenado en ch es a
El carácter almacenado ahora en ch es m
```

Hay que observar en el programa 2.8 que la primera letra almacenada en la variable `ch` es a y la segunda letra almacenada es m. En vista que una variable sólo puede usarse para almacenar un valor a la vez, la asignación de m a la variable causa de manera automática que a se sobrescriba.

Declaraciones múltiples

Las variables que tienen el mismo tipo de datos siempre pueden agruparse y declararse usando una sola instrucción de declaración. La forma común de dicha declaración es

tipo de datos listaDeVariables;

Por ejemplo, las cuatro declaraciones separadas usadas en el programa 2.7,

```
double calif1;
double calif2;
double total;
double promedio;
```

pueden reemplazarse por la instrucción de declaración sencilla

```
double calif1, calif2, total, promedio;
```

Del mismo modo, las dos declaraciones de carácter,

```
char ch;
char clave;
```

pueden reemplazarse con la instrucción de declaración sencilla

```
char ch, clave;
```

Hay que observar que declarar múltiples variables en una sola declaración requiere que el tipo de datos de las variables sólo se dé una vez, que todos los nombres de las variables se separen con comas y que sólo se use un punto y coma para terminar la declaración. El espacio después de cada coma se inserta por legibilidad y no es indipensable.

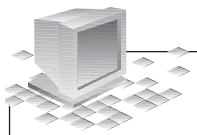
Las instrucciones de declaración también pueden usarse para almacenar un valor en variables declaradas. Por ejemplo, la instrucción de declaración

```
int num1 = 15;
```

declara al mismo tiempo la variable `num1` como una variable entera y establece el valor de 15 en la variable. Cuando se usa una instrucción de declaración para almacenar un valor en una variable, se dice que la variable fue **inicializada**. Por tanto, en este ejemplo es correcto decir que la variable `num1` fue inicializada en 15. Del mismo modo, las instrucciones de declaración

```
double calif1 = 87.0;
double calif2 = 93.5;
double total;
```

declaran tres variables de precisión doble e inicializan dos de ellas. Cuando se usan inicializaciones, la buena práctica de programación dicta que cada variable inicializada sea declarada en una línea individual. Pueden usarse como inicializadores dentro de una función las constantes, las expresiones que usan sólo constantes (como 87.0 + 12 – 2) y expresiones que usan constantes y variables inicializadas con anterioridad. Por ejemplo, el programa 2.7 con una inicialización de declaración se convierte en el programa 2.7a.



Programa 2.7a

```
#include <iostream>
using namespace std;

int main()
{
  double calif1 = 85.5;
  double calif2 = 97.0;
  double total, promedio;

  total = calif1 + calif2;
  promedio = total/2.0;  // divide el total entre 2.0
  cout << "El promedio de las calificaciones es " << promedio << endl;

  return 0;
}
```

Observe la línea en blanco después de la última instrucción de declaración. Insertar una línea en blanco después de las declaraciones de variables colocadas en la parte superior del cuerpo de una función es una buena práctica de programación. Mejora tanto la apariencia de un programa como su legibilidad.

Una característica interesante de C++ es que las declaraciones de variables pueden entremezclarse libremente e incluso estar contenidas con otras instrucciones; el único requisito es que una variable debe declararse antes de usarla. Por ejemplo, la variable `total` en el programa 2.7a podría haberse declarado cuando se usó por primera vez empleando la instrucción double `total = calif1 + calif2`. En situaciones muy restringidas (como en la depuración, como se describe en la sección 3.9, o en una iteración tipo for, descrita en la sección 5.4), puede ser útil declarar una variable en el punto de su primer uso. En general, sin embargo, es preferible no dispersar las declaraciones sino más bien agruparlas en la manera más concisa y clara posible, al principio de cada función.

Asignación de memoria

Las instrucciones de declaración que se han introducido ejecutan tareas tanto de software como de hardware. Desde una perspectiva del software, las instrucciones de declaración siempre proporcionan una lista de todas las variables y sus tipos de datos. En esta función de software, las declaraciones de variable también ayudan a controlar un error común y problemático causado por la escritura equivocada del nombre de una variable dentro de un programa. Por ejemplo, suponga que una variable nombrada `distancia` se declara e inicializa usando la instrucción

```
int distancia = 26;
```

Ahora suponga que esta variable se escribe mal en forma inadvertida en la instrucción

```
mpg = distncia / galones;
```

En lenguajes que no requieren declaraciones de variable, el programa trataría `distancia` como una variable nueva y le asignaría un valor inicial de cero a la variable o usaría cualquier valor que resultara estar en el área de almacenamiento de la variable. En cualquier caso se calcularía un valor y se asignaría a `mpg`, y encontrar el error o siquiera saber que ocurrió podría ser en extremo problemático. Tales errores son imposibles en C++ porque el compilador señalará `distancia` como una variable no declarada. El compilador no puede, por supuesto, detectar cuándo una variable declarada se escribe en lugar de otra variable declarada.

Además de su función en el software, las instrucciones de declaración también pueden realizar una tarea de hardware distinta. Debido a que cada tipo de datos tiene sus propios requerimientos de almacenamiento, la computadora puede asignar suficiente almacenamiento para una variable sólo después de conocer el tipo de datos de ésta. Debido a que las declaraciones de variable proporcionan esta información, pueden usarse para forzar al compilador a reservar almacenamiento suficiente en la memoria física para cada variable. Las instrucciones de declaración usadas con este propósito de hardware se llaman también ***instrucciones de definición*** porque definen o le indican al compilador cuánta memoria es necesaria para el almacenamiento de datos.

Todas las instrucciones de declaración que hemos encontrado hasta ahora han sido instrucciones de definición. Más adelante, se verán casos de instrucciones de declaración que no causan que se asigne ningún almacenamiento nuevo y se usan tan sólo para declarar o alertar al programa de los tipos de datos de variables que se crean en otras partes del programa.

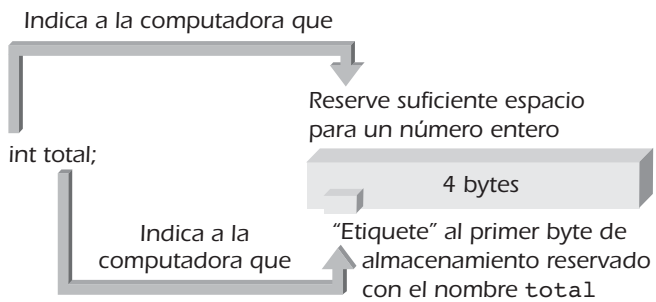


Figura 2.12a *Definición de una variable de entero llamada* `total`.

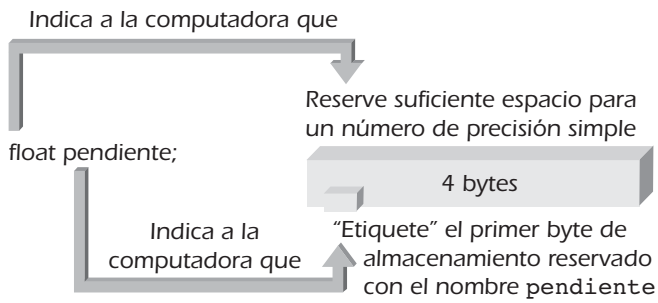


Figura 2.12b *Definición de la variable de punto flotante llamada* `pendiente`.

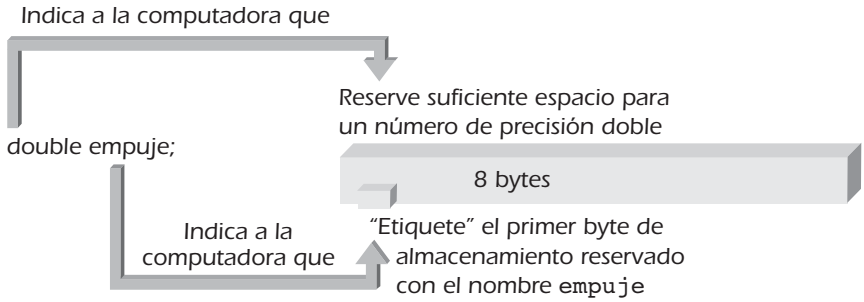


Figura 2.12c *Definición de la variable de precisión doble llamada* `empuje`.

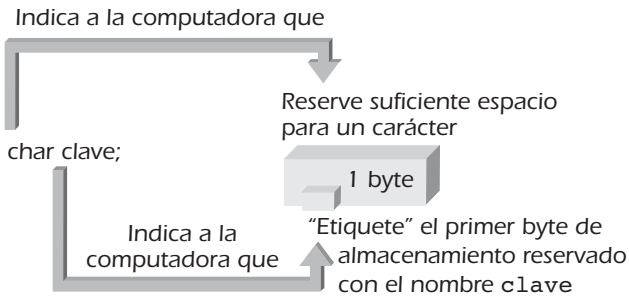


Figura 2.12d *Definición de la variable de carácter llamada* `clave`.

La figura 2.12 (partes a-d) ilustra la serie de operaciones puestas en movimiento por instrucciones de declaración que también desempeñan un papel de definición. La figura muestra que las instrucciones de definición (o, si lo prefiere, instrucciones de declaración que también causan que se asigne memoria) "etiquetan" el primer byte de cada conjunto de bytes reservados con un nombre. Este nombre es, por supuesto, el nombre de la variable y es usado por la computadora para ubicar en forma correcta el punto de inicio del área de memoria reservada de cada variable.

Dentro de un programa, después que se ha declarado una variable, por lo general ésta es usada por el programador para referirse al contenido de la variable (es decir, el valor de la variable). En qué parte de la memoria se almacena este valor comúnmente es de poco interés para el programador. El compilador, sin embargo, debe estar al tanto de dónde se almacena cada valor y ubicar en forma correcta cada variable. En esta tarea la computadora usa el nombre de la variable para ubicar el primer byte de almacenamiento asignado con anterioridad a la variable. Conocer el tipo de datos de la variable le permite luego al compilador almacenar o recuperar el número correcto de bytes.

Despliegue de la dirección de una variable[11]

Cada variable tiene tres elementos importantes asociados a ella: su tipo de datos, el valor real almacenado en la variable y la dirección de la variable. El valor almacenado en la variable se conoce como el contenido de la variable, mientras la dirección de la primera ubicación de

[11]Este tema puede omitirse en la primera lectura sin perder la continuidad.

memoria usada para la variable constituye su dirección. Cuántas ubicaciones se usan en realidad para la variable, como acabamos de ver, depende del tipo de datos de ésta. La relación entre estos tres elementos (tipo, contenido, ubicación) se ilustra en la figura 2.13.

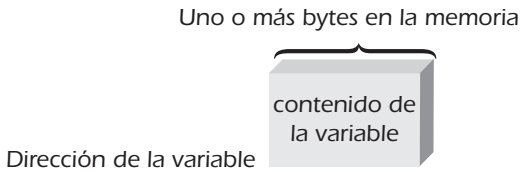


Figura 2.13 *Una variable típica.*

Los programadores por lo general sólo se interesan en el valor asignado a la variable (su contenido) y ponen poca atención al lugar donde está almacenado el valor (su dirección). Por ejemplo, considere el programa 2.9.

Programa 2.9

```
#include <iostream>
using namespace std;

int main()
{
  int num;

  num = 22;
  cout << "El valor almacenado en num es " << num << endl;

  return 0;
}
```

La salida desplegada cuando se ejecuta el programa 2.9 es

```
El valor almacenado en num es 22
```

El programa 2.9 tan sólo imprime el valor 22, el cual es el contenido de la variable `num`. Podemos ir más allá, sin embargo, y preguntar "¿Dónde está almacenado en realidad el número 22?" Aunque la respuesta es "en `num`", ésta es sólo la mitad de la respuesta. El nombre de la variable `num` tan sólo es un símbolo conveniente para ubicaciones físicas reales en la memoria, como se ilustra en la figura 2.14.

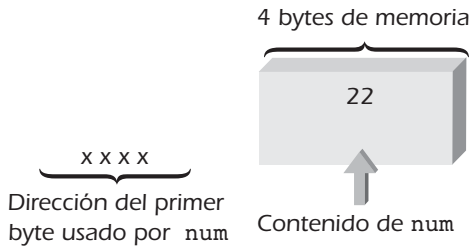


Figura 2.14 *Algún lugar en la memoria.*

Para determinar la dirección de `num`, podemos usar el operador de dirección de C++, `&`, el cual significa "la dirección de". Excepto cuando se usa en una expresión, el operador de dirección colocado enfrente del nombre de una variable se refiere a la dirección de la variable.[12] Por ejemplo, `&num` significa *la dirección de* `num`, `&total` significa *la dirección de* `total` y `&precio` significa *la dirección de* `precio`. El programa 2.10 usa el operador de dirección para mostrar la dirección de la variable `num`.

Programa 2.10

```
#include <iostream>
using namespace std;

int main()
{
  int num;

  num = 22;
  cout << "El valor almacenado en num es " << num << endl;
  cout << "La dirección de num = " << &num << endl;

  return 0;
}
```

La salida del programa 2.10 es

```
El valor almacenado en num es 22
La dirección de num = 0012FED4
```

[12]Cuando se usa para declarar variables y argumentos de referencia, los cuales se presentan en el capítulo 7, el signo & se refiere al tipo de datos que lo *preceden*. Por tanto, la declaración `double &num` se lee como "`num` es la dirección de un `double`" o, de manera más común, como "`num` es una referencia a un `double`"

La figura 2.15 ilustra la información de dirección adicional proporcionada por la salida del programa 2.10.

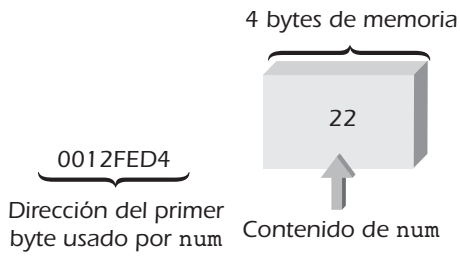


Figura 2.15 *Un panorama más completo de la variable* `num`.

Es evidente que la salida de dirección del programa 2.10 depende de la computadora usada para ejecutar el programa. Sin embargo, cada vez que se ejecuta el programa 2.10 muestra la dirección de la primera ubicación de memoria usada para almacenar la variable `num`. Como lo ilustra la salida del programa 2.10, el despliegue de la dirección está en notación hexadecimal. Este despliegue no tiene efecto en la manera en que se usan las direcciones en forma interna en el programa; tan sólo nos proporciona un medio de desplegar direcciones que es útil para comprenderlas. Como se verá en los capítulos 6 y 12, usar direcciones, en oposición a sólo desplegarlas, es una herramienta de programación importante y poderosa en extremo.

Ejercicios 2.5

1. Establezca si los siguientes nombres de variables son válidos o no. Si son inválidos, explique por qué.

```
prod_a      c1234       abcd        _c3         12345
ampnuevo    vatios      $total      $alnuevo    a1b2c3d4
9ab6        suma.de     promedio    voltios1    finvoltios
```

2. Establezca si los siguientes nombres de variable son válidos o no. Si son inválidos, explique por qué. También indique cuál de los nombres de variable válidos no deberían usarse debido a que no transmiten información sobre la variable.

```
actual      a243        r2d2        primer_num    cc_a1
hector      susana      c3p0        total         suma
maximo      okay        a           increible     veporel
3suma       for         tot.a1      c$cinco       potencianeta
```

3. **a.** Escriba una instrucción de declaración para declarar que se usará la variable `cuenta` para almacenar un entero.
   b. Escriba una instrucción de declaración para declarar que se usará la variable `voltio` para almacenar un número de punto flotante.
   c. Escriba una instrucción de declaración para declarar que se usará la variable `potencia` para almacenar un número de precisión doble.

d. Escriba una instrucción de declaración para declarar que se usará la variable de carácter `clave` para almacenar un carácter.

4. Escriba instrucciones de declaración para las siguientes variables:
- **a.** `num1`, `num2` y `num3` usados para almacenar números enteros
- **b.** `amps1`, `amps2`, `amps3` y `amps4` usados para almacenar números de precisión doble
- **c.** `voltios1`, `voltios2` y `voltios3` usados para almacenar números de precisión doble
- **d.** `codigoA`, `codigoB`, `codigoC`, `codigoD` y `codigoE` usados para almacenar tipos de carácter

5. Escriba instrucciones de declaración para las siguientes variables:
- **a.** `primernum` y `segundonum` usados para almacenar enteros
- **b.** `velocidad`, `aceleración` y `distancia` usados para almacenar números de precisión doble
- **c.** `empuje` usado para almacenar un número de precisión doble

6. Vuelva a escribir cada una de estas instrucciones de declaración como tres declaraciones individuales.
- **a.** `int mes, día = 30, anio;`
- **b.** `double horas, voltios, potencia = 15.62;`
- **c.** `double precio, cantidad, impuestos;`
- **d.** `char teclaEntrada, ch, opción = 'f';`

7. a. Determine el efecto de cada instrucción en el siguiente programa:

```
#include <iostream>
using namespace std;

int main()
{
  int num1, num2, total;
  num1 = 25;
  num2 = 30;
  total = num1 + num2;
  cout << "El total de" << num1 << " y "
       << num2 << " es " << total << endl;

  return 0;
}
```

- **b.** ¿Cuál es la salida que se imprimirá cuando se ejecute el programa mostrado en el ejercicio 7a?

8. Toda variable tiene tres elementos asociados. ¿Cuáles son estos tres elementos?

NOTA PARA LOS EJERCICIOS 9 A 11: Suponga que un carácter requiere un byte de almacenamiento, un entero cuatro bytes, un número de precisión simple cuatro bytes, un número de precisión doble ocho bytes y que a las variables se les asigna almacenamiento en el orden en que son declaradas. (Repase la sección 1.6 si no está familiarizado con el concepto de byte.)

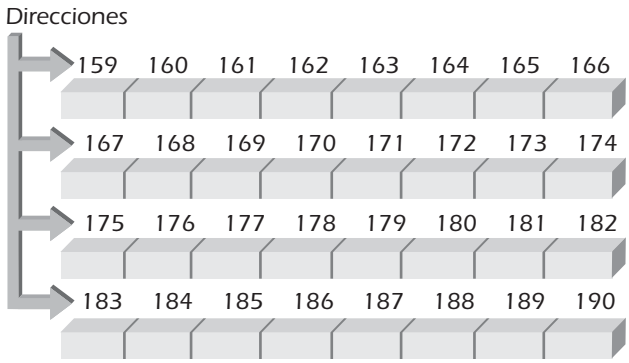


Figura 2.16 *Bytes de memoria para los ejercicios 9, 10 y 11.*

9. a. Usando la figura 2.16 y suponiendo que el nombre de la variable `tasa` se asigna al byte que tiene dirección de memoria 159, determine la dirección correspondiente a cada variable declarada en las siguientes instrucciones. También llene los bytes apropiados con los datos de inicialización incluidos en las instrucciones de declaración. (Use letras para los caracteres, no los códigos de computadora que se almacenarían en realidad.)

```
float tasa;
char ch1 = 'M', ch2 = 'E', ch3 = 'L', ch4 = 'T';
double impuestos;
int num, count = 0;
```

b. Repita el ejercicio 9a, pero sustituya los patrones de byte reales que usaría una computadora que utilice el código ASCII para almacenar los caracteres en las variables `ch1`, `ch2`, `ch3` y `ch4`. (*Sugerencia:* Use el apéndice B.)

10. a. Usando la figura 2.16 y suponiendo que a la variable nombrada `cn1` se le asigna al byte en la dirección de memoria 159, determine las direcciones correspondientes a cada variable declarada en las siguientes instrucciones. Además, llene los bytes apropiados con los datos de inicialización incluidos en las instrucciones de declaración. (Use letras para los caracteres y no los códigos de computadora que se almacenarían en realidad.)

```
char cn1 = 'P', cn2 = 'E', cn3 = 'R', cn4 = 'F', cn5 = 'E';
char cn6 = 'C', cn7 = 'T', key = '\\', sch = '\'', inc = 'A';
char inc1 = 'T';
```

b. Repita el ejercicio 10a, pero sustituya los patrones de byte reales que usaría una computadora que utilice el código ASCII para almacenar los caracteres en cada una de las variables declaradas. (*Sugerencia:* Use la tabla 2.3.)

11. Usando la figura 2.16 y suponiendo que el nombre de variable `millas` se asigna al byte en la dirección de memoria 159, determine las direcciones correspondientes a cada variable declarada en las siguientes instrucciones.

```
float millas;
int cuenta, num;
double dist, temp;
```

2.6 APLICACIÓN DEL PROCEDIMIENTO DE DESARROLLO DE SOFTWARE

Recordará de la sección 1.2 que escribir un programa en C++ es en esencia el tercer paso en el proceso de programación. Los primeros dos pasos en el proceso son determinar lo que se requiere y seleccionar el algoritmo que se va a codificar en C++. En esta sección se muestra cómo los pasos presentados en la sección 1.2 se aplican en la práctica cuando se convierten problemas de programación de trabajo en programas C++. Para repasar, una vez que se establece el requerimiento o problema de un programa, el procedimiento de desarrollo de software consiste de los siguientes pasos:

◆ Paso 1: *Analizar el problema*

El análisis de un problema puede consistir hasta de dos partes. La primera parte es un ***análisis básico*** que debe realizarse en todos los problemas y consiste en extraer la información completa de entrada y salida proporcionada por los problemas. Es decir, debe:

1. Determinar y entender los elementos de salida deseados que debe producir el programa
2. Determinar los elementos de entrada

Juntos, estos dos elementos se conocen como la entrada/salida del problema, I/O (por sus siglas en inglés), para abreviar. Sólo después que se ha determinado la I/O de un problema es posible seleccionar un algoritmo para transformar las entradas en las salidas deseadas. En este punto, en ocasiones es necesario o útil, o ambas cosas, realizar un cálculo manual para verificar que en efecto puede obtenerse la salida a partir de las entradas. Es claro que si se ha proporcionado una fórmula que relaciona las entradas con las salidas, este paso puede omitirse en esta etapa. Si las entradas requeridas están disponibles y la salida o salidas deseadas pueden producirse, se dice que el problema está definido con claridad y puede resolverse.

Por una variedad de razones puede no ser posible completar un análisis básico. Si esto es así, puede ser necesario un análisis extendido. Un ***análisis extendido*** tan sólo significa que debe obtener información adicional sobre el problema, de modo que pueda entender a fondo lo que se está pidiendo y cómo lograr el resultado. En este texto cualquier información adicional requerida para una comprensión del problema se suministrará junto con el planteamiento de éste.

◆ Paso 2: *Desarrollar una solución*

Este paso con frecuencia se conoce como paso de diseño, y usaremos los términos ***diseño*** y ***desarrollo*** de forma indistinta. En este paso debe elegir un algoritmo para transformar los elementos de entrada en las salidas deseadas y refinarlo según sea necesario para definir de manera adecuada todas las características que desea que tenga el programa. Si no ha realizado un cálculo manual usando el algoritmo en el paso de análisis, debería hacerlo ahora, usando valores de entrada específicos.

Al diseñar una solución, el enfoque específico que adoptaremos se conoce a menudo como ***enfoque descendente***. Este enfoque consiste en comenzar con la solución más general y refinarla de tal manera que la solución final consista en tareas definidas con claridad que puedan ser completadas por funciones individuales del programa.

◆ Paso 3: Codificar la solución

En este punto se escribe en realidad el programa de C++ que corresponde a la solución desarrollada en el paso 2.

◆ Paso 4: Prueba y corrección del programa

Esto se realiza por medio de datos de prueba seleccionados y se utliza para hacer correcciones al programa cuando se encuentran errores. Un conjunto de datos de prueba que siempre debería usarse son los datos utilizados en su cálculo manual previo.

Para ver cómo pueden aplicarse cada uno de estos pasos en la práctica, ahora los empleamos en el siguiente problema de programación simple.

La resistencia eléctrica, r*, de un alambre metálico, en ohmios, está dada por la fórmula* r = (ml)/a*, donde* m *es la resistividad del metal;* l *es el largo del alambre, en pies; y* a *es el área de corte transversal del alambre, en circular mils. Usando esta información, escriba un programa en C++ para calcular la resistencia de un alambre que mide 125 pies de largo, tiene un área de corte transversal de 500 mils circulares y es de cobre. La resistividad del cobre,* m*, es 10.4.*

Paso 1 Analizar el problema

El primer paso para desarrollar un programa para este problema planteado es realizar un análisis básico. Comenzaremos por determinar las salidas requeridas. Con frecuencia, en el planteamiento del problema usarán palabras como *calcular*, *imprimir*, *determinar*, *encontrar* o *comparar*, las cuales pueden emplearse para determinar las salidas deseadas.

Para el planteamiento del problema de muestra, el enunciado clave es "calcular la resistencia de un alambre". Esto identifica con claridad un elemento de salida. Debido a que no hay otros enunciados así en el problema, sólo se requiere una salida.

Después que se ha identificado con claridad la salida deseada, el paso de análisis básico continúa con la identificación de todos los elementos de entrada. Es esencial en esta etapa distinguir entre elementos de entrada y valores de entrada. Un **elemento de entrada** es el nombre de una cantidad de entrada, mientras un **valor de entrada** es un número o cantidad específica que puede ser el elemento de entrada. Por ejemplo, en el planteamiento del problema de muestra, los elementos de entrada son la resistividad, m, el largo del alambre, l, y el área de corte transversal del alambre, a. Aunque estos elementos de entrada tienen valores numéricos específicos, estos valores de los elementos de entrada por lo general no son de importancia en esta etapa.

La razón por la que los valores de entrada no son necesarios en este punto es que la selección de un algoritmo por lo general es independiente de valores de entrada específicos. El algoritmo depende de saber cuáles son los elementos de salida y entrada y si hay algunos límites especiales. Veamos por qué esto es así.

Del planteamiento del problema es claro que el algoritmo para transformar los elementos de entrada en la salida deseada está dado por la fórmula $r = (ml)/a$. Hay que observar que esta fórmula puede usarse sin tener en cuenta los valores específicos asignados a m, l o a. Aunque no se puede producir un valor numérico real para el elemento de salida (resistencia) a menos que se tengan valores numéricos reales para el elemento de entrada, la relación correcta entre entradas y salidas está expresada por la fórmula. Recuerde que esto es precisamente lo que proporciona un algoritmo: una descripción de cómo las entradas se transforman en salidas que funcione para todas las entradas.

Paso 2 Desarrollar una solución

El algoritmo básico para transformar las entradas en la salida deseada es proporcionado por la fórmula dada. Ahora debe refinarse enumerando, con detalle, cómo se han de combinar las entradas, las salidas y el algoritmo para producir una solución. Este listado indica los pasos que seguirá el programa para resolver el problema. Como tal constituye un esbozo de la forma final que seguirá el código del programa. Usando seudocódigo, el algoritmo completo para resolver este problema es

Asignar valores a m, l y a
Calcular la resistencia usando la fórmula r = (ml)/a
Mostrar la resistencia

Hay que observar que la estructura de este algoritmo se ajusta a la estructura de control secuencial presentada en la sección 1.2.

Habiendo seleccionado y depurado el algoritmo, el siguiente paso en el diseño (si no se hizo ya en el paso de análisis) es comprobar el algoritmo en forma manual usando datos específicos. Realizar un cálculo manual, ya sea con papel y lápiz o por medio de una calculadora, ayuda a verificar que en realidad ha entendido el problema. Una ventaja adicional de hacer un cálculo manual es que los resultados pueden usarse después para compararlos con la operación del programa en la fase de prueba. Entonces, cuando el programa final se use con otros datos, habrá establecido un grado de confianza en que se está calculando un resultado correcto.

Hacer un cálculo manual requiere que se tengan valores de entrada específicos que puedan ser asignados y usados por el algoritmo para producir la salida deseada. Para este problema se dan tres valores de entrada: una resistividad de 10.4, un área de corte transversal de 500 circular mils y un largo de 125 pies. Al sustituir estos valores en la fórmula, se obtiene una resistencia = (10.4)(125)/500 = 2.60 ohmios para el alambre de cobre.

Paso 3 Codificar la solución

Debido a que se ha desarrollado en forma minuciosa una solución de programa, todo lo que resta es codificar el algoritmo de solución en C++. Esto significa declarar variables de entrada y salida apropiadas, inicializar las variables de entrada en forma correcta, calcular la resistencia e imprimir el valor de resistencia calculado. El programa 2.11 ejecuta estos pasos.

Programa 2.11

```
#include <iostream>
using namespace std;
int main()
{
  double resistividad, area, longitud, resistencia;

  resistividad = 10.4;
  area = 500;
  longitud = 125;
  resistencia = (resistividad * longitud) / area;

  cout << "La resistencia del alambre (en ohmios) es "
          << resistencia << endl;

  return 0;
}
```

Cuando se ejecuta el programa 2.11, se produce la siguiente salida:

```
La resistencia del alambre (en ohmios) es 2.6
```

Ahora que se tiene un programa funcional que produce un resultado, puede comenzar el paso final en el proceso de desarrollo, probar el programa.

Paso 4 Prueba y corrección del programa

El propósito de probar un programa es verificar que funciona en forma correcta y en realidad satisface los requerimientos. Una vez que se ha completado la prueba, el programa puede usarse para calcular salidas para diferentes datos de entrada sin necesidad de volver a probarlo. Éste es, por supuesto, uno de los valores reales de escribir un programa; el mismo programa puede usarse una y otra vez con datos de entrada nuevos.

El método de prueba más simple es verificar la operación del programa con conjuntos de datos de entrada seleccionados cuidadosamente. Un conjunto de datos de entrada que siempre debería usarse son los datos que se seleccionaron para el cálculo manual realizado con anterioridad en el paso 2 del procedimiento de desarrollo. En este caso el programa es relativamente simple y sólo ejecuta un cálculo. Debido a que la salida producida por la ejecución de prueba coincide con el cálculo manual se tiene un buen grado de confianza de que puede ser usado para calcular correctamente la resistencia de otros valores de entrada.

Ejercicios 2.6

NOTA: En cada uno de estos ejercicios se expone un problema de programación. Lea el planteamiento del problema primero y luego responda las preguntas relacionadas con él.

No *escriba un programa para resolver los problemas, sino tan sólo responda las preguntas que siguen a la especificación del programa.*

1. Suponga que tiene que escribir un programa en C++ para calcular la resistencia total de un circuito en serie. En dicho circuito la resistencia total es la suma de todos los valores de resistencia individuales. Suponga que el circuito consiste en una cantidad de resistores de 56 ohmios, 33 ohmios y 15 ohmios.
   a. Para este problema de programación, ¿cuántas salidas se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida. Suponga que la cantidad de resistores de 56 ohmios es *m*, la cantidad de resistores de 33 ohmios es *n*, la cantidad de resistores de 15 ohmios es *p*.
   d. Pruebe el algoritmo escrito para la parte c usando la siguiente muestra de datos: $m = 17$, $n = 24$ y $p = 12$.

2. Suponga que tiene que escribir un programa para calcular el valor de la *distancia*, en millas, dada la relación:

   $$distancia = velocidad * tiempo\ transcurrido$$

   a. Para este problema de programación, ¿cuántas salidas se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
   d. Pruebe el algoritmo escrito para la parte c usando la siguiente muestra de datos: *velocidad* es 55 millas por hora y *tiempo transcurrido* es 2.5 horas.
   e. ¿Cómo debe modificarse el algoritmo que determinó en la parte c si el tiempo transcurrido se diera en minutos en lugar de horas?

3. Suponga que tiene que escribir un programa para determinar el valor de *ergios*, dadas las relaciones:

   $$Ergios = fergios * \sqrt{lergios}$$
   $$Lergios = 2 * \pi * e^{\mu}$$

   a. Para este problema de programación, ¿cuántas salidas se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
   d. Pruebe el algoritmo escrito para la parte c usando la siguiente muestra de datos: $fergios = 14.65$, $\pi = 3.1416$, $\mu = 1.672$ y $e = 2.7818$.

4. Suponga que tiene que escribir un programa para mostrar las siguientes especificaciones:

Amplificación de voltaje:	35
Potencia de salida:	2.5 vatios
Ancho de banda:	15KHz

   a. Para este problema de programación, ¿cuántas líneas de salida se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.

5. Escriba un programa en C++ para determinar la distancia recorrida por un automóvil después de 10 segundos, suponiendo que el automóvil viaja inicialmente a 60 millas por hora y el conductor aplica los frenos para desacelerar de manera uniforme a una velocidad de 12 millas/s^2. Use el hecho que *distancia* = $s - (1/2)dt^2$, donde s es la velocidad inicial del automóvil, d es la desaceleración y t es el tiempo transcurrido.
   a. Para este problema de programación, ¿cuántas salidas se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
   d. Pruebe el algoritmo escrito para la parte c usando los datos proporcionados en el problema.

6. Considere el siguiente problema de programación: en 1627, la isla de Manhattan fue vendida a los colonizadores holandeses por aproximadamente 24 dólares. Si las ganancias de esa venta se hubieran depositado en un banco holandés que pagara 5% de interés anual compuesto, ¿cuál sería el saldo principal al final de 2002? Se requiere un despliegue como sigue: El saldo al 31 de diciembre de 2002 es: xxxxxx, donde xxxxxx es la cantidad calculada por su programa.
   a. Para este problema de programación, ¿cuántas salidas se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
   d. Pruebe el algoritmo escrito para la parte c usando los datos proporcionados en el planteamiento del problema.

7. Escriba un programa que calcule y despliegue los voltajes de salida de dos circuitos eléctricos y la suma de los dos voltajes. El voltaje de salida para el primer circuito está dado por la ecuación (150) $V / 0.38f$ y el voltaje de salida para el segundo circuito está dado por la ecuación

$$\frac{230V}{\sqrt{56^2 + (0.98f)^2}}$$

   donde V es el voltaje de entrada al circuito y f es la frecuencia en Hertz.
   a. Para este problema de programación, ¿cuántas salidas se requieren?
   b. ¿Cuántas entradas tiene este problema?
   c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
   d. Pruebe el algoritmo escrito para la parte c usando la siguiente muestra de datos: el primer circuito es operado con un voltaje de entrada de 1.2 voltios a una frecuencia de 144 Hertz y el segundo circuito es operado con un voltaje de entrada de 2.3 voltios a 100 Hertz.

8. Considere el siguiente problema de programación: la fórmula para la desviación normal estándar, z, usada en aplicaciones estadísticas es

$$z = \frac{X - \mu}{\sigma}$$

   donde μ se refiere a un valor medio y σ a una desviación estándar. Usando esta fórmula, escriba un programa que calcula y despliega el valor de la desviación normal estándar cuando $X = 85.3$, $\mu = 80$ y $\sigma = 4$.

a. Para este problema de programación, ¿cuántas salidas se requieren?
b. ¿Cuántas entradas tiene este problema?
c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
d. Pruebe el algoritmo escrito para la parte c usando los datos proporcionados en el problema.

9. La ecuación de la curva normal (en forma de campana) usada en aplicaciones estadísticas es

$$y = \frac{1}{\sigma\sqrt{2\pi}} e^{-(1/2)[(x-\mu)/\sigma]^2}$$

Usando esta ecuación, suponga que tiene que escribir un programa en C++ que calcule el valor de y.
a. Para este problema de programación, ¿cuántas salidas se requieren?
b. ¿Cuántas entradas tiene este problema?
c. Determine un algoritmo para convertir los elementos de entrada en elementos de salida.
d. Pruebe el algoritmo escrito para la parte c suponiendo que $\mu = 90$, $\sigma = 4$, $x = 80$ y $\pi = 3.1416$.

2.7 APLICACIONES

En esta sección, el procedimiento de desarrollo de software presentado en la sección anterior se aplica a dos problemas de programación específicos. Aunque cada problema es diferente, el procedimiento de desarrollo funciona para ambas situaciones. Este procedimiento puede aplicarse a cualquier problema de programación para producir un programa completo y forma el fundamento para todos los programas desarrollados en este texto.

Aplicación 1: Trampas de un radar de velocidad

Un radar común de detección de velocidad de la policía de caminos emite un rayo de microondas a una frecuencia f_0. El rayo es reflejado por un automóvil que se aproxima y el rayo reflejado es captado y analizado por la unidad de radar. La frecuencia del rayo reflejado es cambiada ligeramente de f_0 a f_1 debido al movimiento del automóvil. La relación entre la velocidad del automóvil, v, en millas por hora, y las dos frecuencias de microondas es

$$v = (6.685 \times 10^8)(f_1 - f_0) / (f_1 + f_0)$$

donde las ondas emitidas tienen una frecuencia de $f_0 = 2 \times 10^{10}$ sec^{-1}. Usando la fórmula dada, escriba un programa en C++ para calcular y desplegar la velocidad correspondiente a una frecuencia recibida de 2.000004×10^{10} sec^{-1}.

Ahora aplicamos el procedimiento de desarrollo de software a este problema.

Paso 1 *Analizar el problema*

Para este problema el programa requiere una sola salida: la velocidad del automóvil. Los elementos de entrada requeridos para solucionar la velocidad son la frecuencia emitida, f_0, y la frecuencia recibida, f_1.

Paso 2 Desarrollar una solución

El algoritmo proporcionado para transformar los tres elementos de entrada en el elemento de salida deseado está dado por la fórmula $v = 6.685 \times 10^8(f_1 - f_0)/(f_1 + f_0)$. Por tanto, el algoritmo completo para el programa de solución es

Asignar valores a f_1 y f_0
Calcular la velocidad usando la fórmula $v = 6.685 \times 10^8(f_1 - f_0)/(f_1 + f_0)$
Desplegar la velocidad

Un cálculo manual, usando los datos $f_0 = 2 \times 10^{10}$ sec^{-1} y $f_1 = 2.000004 \times 10^{10}$ sec^{-1} produce una velocidad de 66.85 millas por hora.

Paso 3 Codificar la solución

El programa 2.12 proporciona el código necesario.

Programa 2.12

```
#include <iostream>
using namespace std;

int main()
{
  double velocidad, f0, f1;

  f0 = 2e-10;
  f1 = 2.0000004e-10;

  velocidad = 6.685e8 * (f1 - f0) / (f1 + f0);
  cout << "La velocidad es " << velocidad << "millas/hora " <<endl;

  return 0;
}
```

El programa 2.12 comienza con un comando preprocesador `#include` seguido por una función `main()`. Esta función comienza con la palabra clave `main` y termina con la llave de cierre, }. Además, el programa 2.12 contiene una instrucción de declaración, tres instrucciones de asignación y una instrucción de salida. Las instrucciones de asignación `f0 = 2e-10` y `f1 = 2.0000004e-10` se usan para inicializar las variables `f0` y `f1` respectivamente. La instrucción de asignación

```
velocidad = 6.685e8 * (f1 - f0) / (f1 + f0);
```

calcula un valor para la variable denominada `velocidad`. Cuando el programa 2.12 es compilado y ejecutado se produce la siguiente salida.

```
La velocidad es 66.85 millas/hora
```

Paso 4 Probar y corregir el programa

El último paso en el procedimiento de desarrollo es probar la salida. Debido a que el cálculo único y el valor desplegado concuerdan con el cálculo manual anterior, se ha verificado la operación correcta del programa. Esto permite usar el programa para valores diferentes de frecuencias recibidas. Hay que observar que si los paréntesis no estuvieran colocados en forma correcta en la instrucción de asignación que calculó un valor para la velocidad, el valor desplegado no concordaría con nuestro cálculo manual previo. Esto nos habría alertado del hecho que hubo un error en el programa.

Aplicación 2: Redes de conmutación telefónica

Una red telefónica conectada en forma directa es aquella en la que todos los teléfonos en la red están conectados en forma directa y no requieren una estación de conmutación central para establecer llamadas entre dos de ellos. Por ejemplo, las instituciones financieras en Wall Street usan una red así para mantener líneas telefónicas abiertas en forma directa y continua entre las empresas.

El número de líneas directas necesarias para mantener una red conectada en forma directa para n teléfonos está dado por la fórmula:

$$líneas = n(n - 1)/2$$

Por ejemplo, conectar en forma directa cuatro teléfonos requiere 6 líneas individuales (véase la figura 2.17). Agregar un quinto teléfono a la red ilustrada en la figura 2.17 requeriría 4 líneas adicionales para un total de 10 líneas.

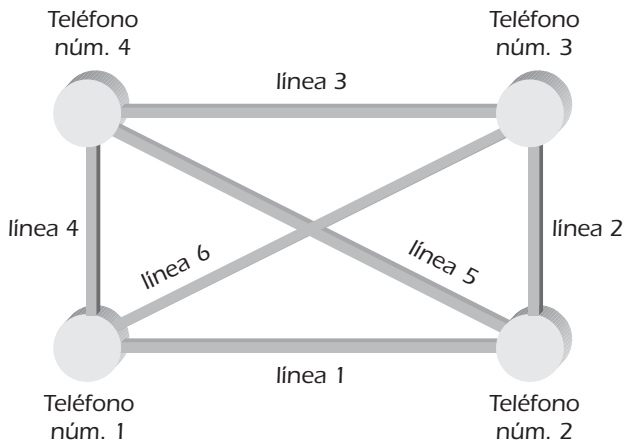


Figura 2.17 Conexión en forma directa de cuatro teléfonos.

Usando la fórmula dada, escriba un programa en C++ que determine el número de líneas directas requeridas para 100 teléfonos, y las líneas adicionales requeridas si se fueran a agregar 10 teléfonos nuevos a la red.

Paso 1 Analizar el problema

Para este programa se requieren dos salidas: el número de líneas directas para 100 teléfonos y el número de líneas adicionales necesarias cuando se agregan 10 teléfonos nuevos a la red existente. El elemento de entrada requerido para este problema es el número de teléfonos, el cual se denota como *n* en la fórmula.

Paso 2 Desarrollar una solución

La primera salida se obtiene con facilidad usando la fórmula:

$líneas = n(n - 1)/2$

Aunque no se proporciona una fórmula para las líneas adicionales, puede usarse la fórmula dada para determinar el número total de líneas necesarias para 110 suscriptores. Restar el número de líneas para 100 suscriptores del número de líneas necesarias para 110 suscriptores producirá entonces el número de líneas adicionales requeridas. Por tanto, el algoritmo completo para el programa, en seudocódigo, es

Calcular el número de líneas directas para 100 suscriptores
Calcular el número de líneas directas para 110 suscriptores
Calcular las líneas adicionales necesarias, que son la diferencia entre el segundo y el primer cálculo.
Desplegar el número de líneas para 100 suscriptores
Desplegar las líneas adicionales necesarias

Verificar este algoritmo en forma manual, usando los datos proporcionados, produce la respuesta:

$líneas = 100(100 - 1)/2 = 100(99)/2 = 4950$

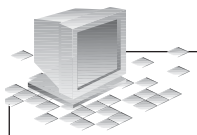
para 100 teléfonos y

$líneas = 5995$

para 110 teléfonos. Por tanto, serían necesarias 1045 líneas adicionales para conectar en forma directa los diez teléfonos adicionales en la red existente.

Paso 3 Codificar la solución

El programa 2.13 proporciona el código necesario.



Programa 2.13

```
#include <iostream>
using namespace std;

int main()
{
  int numin1, numin2, lineas1, lineas2;

  numin1 = 100;
  numin2 = 110;
  lineas1 = numin1 * (numin1 - 1)/2;
  lineas2 = numin2 * (numin2 - 1)/2;
  cout << "El número de líneas iniciales es " << lineas1 << "." << endl;
  cout << "Se necesitan " << lineas2 - lineas1
       << " líneas adicionales." << endl;

  return 0;
}
```

Como antes, el programa en C++ incluye el archivo de encabezado `iostream` y consiste en una función `main()`. El cuerpo de esta función comienza con la llave de apertura {, y termina con la llave de cierre}. Debido a que el número de líneas entre suscriptores debe ser un número entero (no es posible una línea fraccionaria) las variables `lineas1` y `lineas2` se especifican como variables enteras. Las primeras dos instrucciones de asignación inicializan las variables `numin1` y `numin2`. La siguiente instrucción de asignación calcula el número de líneas necesarias para 100 suscriptores y la última instrucción de asignación calcula el número de líneas para 110 suscriptores. La primera instrucción `cout` se usa para desplegar un mensaje y el resultado del primer cálculo. La siguiente instrucción `cout` se usa para desplegar la diferencia entre los dos cálculos. La siguiente salida se produce cuando se compila y ejecuta el programa 2.13:

```
El número de líneas iniciales es 4950.
Se necesitan 1045 líneas adicionales.
```

Paso 4 *Probar y corregir el programa*

Como los dos cálculos y los valores desplegados concuerdan con el cálculo manual previo, se ha verificado la operación correcta del programa.

Ejercicios 2.7

1. a. Modifique el programa 2.12 para calcular la velocidad de un automóvil cuya frecuencia de retorno del radar es $2.00000035 \times 10^{10}$ sec^{-1}.

b. Compile y ejecute el programa escrito para el ejercicio 1a en una computadora.

2. a. Modifique el programa 2.12 para determinar la frecuencia que será regresada por un automóvil que viaja a 55 millas por hora. Su programa deberá producir el siguiente despliegue:

```
La frecuencia regresada correspondiente a 55 millas por hora
es_____
```

donde el subrayado es reemplazado por el valor real calculado por su programa.

b. Compile y ejecute el programa escrito para el ejercicio 2a en una computadora. Asegúrese de hacer un cálculo manual de modo que pueda verificar los resultados producidos por su programa.

c. Después de haber verificado los resultados del programa escrito en ejercicio 2a, modifique el programa para calcular la frecuencia de vuelta para un automóvil que viaja a 75 millas por hora.

3. a. Modifique el programa 2.13 para calcular y desplegar el número total de líneas necesario para conectar 1000 teléfonos individuales en forma directa entre sí.

b. Compile y ejecute el programa escrito para el ejercicio 3a en una computadora.

4. a. Modifique el programa 2.13 de modo que una nueva variable `numfin`, el número adicional de suscriptores que va a ser conectados a la red existente, se inicialice en 10. Haga otros cambios en el programa de modo que produzca el mismo despliegue que el programa 2.13.

b. Compile y ejecute el programa escrito para el ejercicio 4a en una computadora. Verifique que el despliegue producido por su programa corresponde al despliegue mostrado en el texto.

5. a. Diseñe, escriba, compile y ejecute un programa en C++ para convertir temperatura en grados Fahrenheit en grados Celsius. La ecuación para esta conversión es

Celsius = 5.0/9.0 (*Fahrenheit* - 32.0).

Haga que su programa convierta y despliegue la temperatura Celsius correspondiente a 98.6 grados Fahrenheit. Su programa deberá producir el siguiente despliegue:

```
Para una temperatura Fahrenheit de_______ grados,
  la temperatura Celsius equivalente es_______ grados.
```

donde su programa debe insertar los valores apropiados en lugar de los subrayados.

b. Verifique en forma manual los valores calculados por su programa. Después que haya comprobado que su programa funciona en forma correcta, modifíquelo para convertir 86.5 grados Fahrenheit en su valor Celsius equivalente.

6. a. Escriba, compile y ejecute un programa en C++ para calcular la resistencia de un circuito en serie consistente en 12 resistores de 56 ohmios, veinte de 39 ohmios, 32 de 27 ohmios y 27 de 15 ohmios. Use el hecho que la resistencia de un circuito en serie es la suma de todas las resistencias individuales. Su programa deberá producir el siguiente despliegue:

```
La resistencia total, en ohmios, es xxxx
```

donde xxxx es reemplazado por el valor de resistencia real calculado por su programa.

b. Verifique en forma manual los valores calculados por su programa. Después que haya comprobado que su programa funciona en forma correcta, modifíquelo para calcular la resistencia de un circuito en serie consistente en ningún resistor de 56 ohmios, 17 de 39 ohmios, 19 de 27 ohmios y 42 de 15 ohmios.

7. a. Diseñe, escriba, compile y ejecute un programa en C++ para calcular el tiempo requerido para hacer un viaje de 183.67 millas. La ecuación para calcular el tiempo transcurrido es

tiempo transcurrido = distancia total / velocidad promedio

Suponga que la velocidad promedio durante el viaje fue 58 millas por hora.

b. Verifique en forma manual los valores calculados por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para determinar el tiempo que toma hacer un viaje de 372 millas a una velocidad promedio de 67 millas por hora.

8. a. Diseñe, escriba, compile y ejecute un programa en C++ para calcular la suma de los números del 1 al 100. La fórmula para calcular esta suma es

$$suma = (n/2) (2*a + (n - 1)d)$$

donde n = número de términos que se van a sumar, a = el primer número y d = la diferencia entre cada número.

b. Verifique en forma manual los valores calculados por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para determinar la suma de los enteros de 100 a 1000.

NOTA: Los ejercicios 9 a 13 requieren elevar un número a una potencia. Esto puede lograrse usando la función de potencia de C++ `pow()`*. Por ejemplo, la instrucción* `pow(2.0,5.0);` *eleva el número 2.0 a la quinta potencia, y la instrucción* `pow(num1,num2);` *eleva la variable* `num1` *a la potencia* `num2`*. Para usar la función de potencia se coloca un comando preprocesador* `#include <cmath>` *en una sola línea después del comando* `#include <iostream>` *o se incluye la instrucción de declaración* `double pow();` *con las instrucciones de declaración de variables usadas en su programa. La función de potencia se explicará con más detalle en la sección 3.3.*

9. a. La ley de Newton del enfriamiento establece que cuando un objeto con una temperatura inicial T se introduce en una sustancia de temperatura A, alcanzará una temperatura *TFIN* en t minutos de acuerdo con la

$$TFIN = (T - A)\, e^{-kt} + A$$

En esta fórmula e es el número irracional 2.71828 redondeado a cinco lugares decimales, conocido por lo común como el número de Euler, y k es un coeficiente térmico, el cual depende del material que se va a enfriar. Usando esta fórmula escriba, compile y ejecute un programa en C++ que determine la temperatura alcanzada por un objeto después de 20 minutos cuando es colocado en un vaso con agua cuya temperatura es de 60 grados. Suponga que el objeto tenía inicialmente una temperatura de 150 grados y tiene una constante térmica de 0.0367.

b. Verifique en forma manual el valor calculado por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para determinar la temperatura alcanzada después de 10 minutos cuando es colocado en un vaso con agua cuya temperatura es de 50 grados.

10. a. La ganancia de voltaje de un amplificador está dada por la fórmula

$$\textit{ganancia de voltaje} = [275 / (23^2 + (0.5f)^2)^{1/2}]^n$$

donde f es la frecuencia, en Hertz, y n es el número de etapas en el amplificador. Usando esta fórmula escriba, compile y ejecute un programa en C++ que determine el valor de la ganancia de voltaje para un amplificador de cuatro etapas que opera a una frecuencia de 120 Hertz. Su programa deberá producir el siguiente despliegue:

```
A una frecuencia de xxxxx hertz, la ganancia de voltaje
es yyyyy
```

donde xxxxx es reemplazado por la frecuencia e yyyyy por la ganancia de voltaje.

b. Verifique en forma manual el valor calculado por su programa. Después que haya comprobado que su programa funciona en forma correcta, modifíquelo para determinar la ganancia de voltaje de un amplificador de 12 etapas que opera a una frecuencia de 9500 Hertz.

11. a. La corriente eléctrica, i, en amperios, que fluye a través del circuito ilustrado en la figura 2.18 está dada por la

$$i = \frac{E(1-e)^{-(R/L)t}}{R}$$

donde E es el voltaje de la batería en voltios, R es el valor del resistor en ohmios, L es el valor del inductor en henrios, t es el tiempo en segundos después de cerrar el interruptor y e es el número de Euler, el cual es 2.718 con una precisión de tres cifras decimales. Usando esta fórmula, escriba, compile y ejecute un programa en C++ para determinar el flujo de corriente en el circuito ilustrado en la figura 2.18 cuando t es 0.12 segundos.

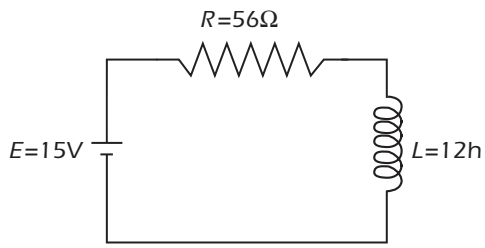


Figura 2.18 Un circuito RL en serie.

b. Verifique en forma manual el valor calculado por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para determinar la corriente en 0.12 segundos si E es de 25 voltios, R es de 33 ohmios y L es de 15 henrios.

12. a. La corriente eléctrica, i, en amperios, que fluye a través del circuito ilustrado en la figura 2.19 está dado por la siguiente ecuación:

$$i = \frac{(E)e^{-t/RC}}{R}$$

donde E es el voltaje de la batería en voltios, R es el valor del resistor en ohmios, C es el valor del capacitor en faradios, t es el tiempo en segundos después de cerrar el interruptor y e es el número de Euler, el cual es 2.718 con una precisión de tres cifras decimales. Usando esta fórmula, escriba, compile y ejecute un programa en C++ para determinar el voltaje a través del capacitor ilustrado en la figura 2.19 cuando t es 0.31 segundos.

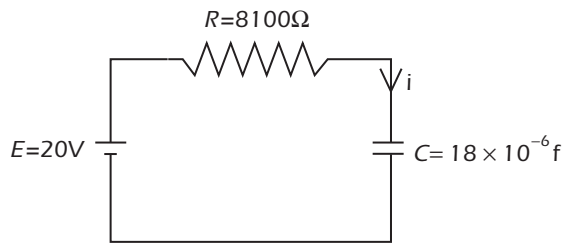


Figura 2.19 Un circuito RC en serie.

b. Verifique en forma manual el valor calculado por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para determinar la corriente en 0.85 segundos si E es de 25 voltios, R es de 220 ohmios y C es de 0.00039 faradios.

13. a. El voltaje eléctrico, V, en voltios, a través del capacitor, C, ilustrado en la figura 2.20 está dado por la ecuación

$$V = \frac{E\left[1 - e^{-t/RC}\right]}{R}$$

donde E es el voltaje de la batería en voltios, R es el valor del resistor en ohmios, C es el valor del capacitor en faradios, t es el tiempo en segundos después de cerrar el interruptor y e es el número de Euler, el cual es 2.718 con una precisión de tres cifras decimales. Usando esta fórmula, escriba, compile y ejecute un programa en C++ para determinar el voltaje a través del capacitor ilustrado en la figura 2.20 cuando t es 0.42 segundos.

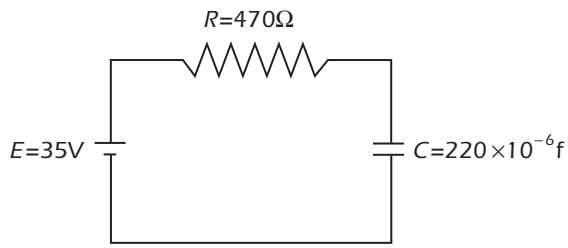


Figura 2.20 Un circuito RC en serie.

b. Verifique en forma manual el valor calculado por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para determinar la corriente en 0.85 segundos si E es de 25 voltios, R es de 220 ohmios y C es de 0.00039 faradios.

14. a. El conjunto de ecuaciones lineales

$$a_{11}X_1 + a_{12}X_2 = c_1$$
$$a_{21}X_1 + a_{22}X_2 = c_2$$

puede resolverse usando la regla de Cramer:

$$X_1 = \frac{c_1 a_{22} - c_2 a_{12}}{a_{11} a_{22} - a_{12} a_{21}}$$
$$X_2 = \frac{c_2 a_{11} - c_1 a_{21}}{a_{11} a_{22} - a_{12} a_{21}}$$

Usando estas ecuaciones, escriba, compile y ejecute un programa en C++ para encontrar los valores X_1 y X_2 que satisfagan las siguientes ecuaciones:

$$3X_1 + 4X_2 = 40$$
$$5X_1 + 2X_2 = 34$$

b. Verifique en forma manual los valores calculados por su programa. Después de haber comprobado que su programa funciona en forma correcta, modifíquelo para resolver el siguiente conjunto de ecuaciones:

$$3X_1 + 12.5X_2 = 22.5$$
$$4.2X_1 - 6.3X_2 = 30$$

2.8 ERRORES COMUNES DE PROGRAMACIÓN

Parte de aprender cualquier lenguaje de programación es cometer los errores elementales que se encuentran por lo común cuando se empieza a usar el lenguaje. Estos errores tienden a ser bastante frustrantes debido a que cada lenguaje tiene su propio conjunto de errores de programación comunes esperando a los incautos. Los errores más comunes cometidos cuando se empieza a programar en C++ incluyen los siguientes.

1. Omitir los paréntesis después de `main`.
2. Omitir o escribir de manera incorrecta la llave de apertura `{` que indica el inicio de un cuerpo de función.
3. Omitir o escribir de manera incorrecta la llave de cierre `}` que indica el final de una función.
4. Escribir mal el nombre de un objeto o función; por ejemplo, escribir `cot` en lugar de `cout`.
5. Olvidar cerrar una cadena enviada a `cout` con un símbolo de comillas.
6. Olvidar separar flujos de datos individuales pasados a `cout` con un símbolo de inserción ("enviar a"), `<<`.
7. Omitir el punto y coma al final de cada instrucción de C++.

8. Agregar un punto y coma al final del comando preprocesador `#include`.
9. Olvidar `\n` para indicar una línea nueva.
10. Escribir en forma incorrecta la letra O en lugar del número cero (0), o viceversa. Escribir de forma incorrecta la letra l, por el número 1, o viceversa.
11. Olvidar declarar todas las variables usadas en el programa. Este error es detectado por el compilador y se genera un mensaje de error para todas las variables no declaradas.
12. Almacenar un tipo de datos inapropiado en una variable declarada. Este error es detectado por el compilador y el valor asignado es convertido al tipo de datos de la variable a la que fue asignado.
13. Usar una variable en una expresión antes que se haya asignado un valor a la variable. Aquí, cualquier valor que resulte estar en la variable se usará cuando se evalúa la expresión, y el resultado carecerá de significado.
14. Dividir valores enteros en forma incorrecta. Este error se disfraza por lo general dentro de una expresión más grande y puede ser un error muy problemático de detectar. Por ejemplo, la expresión

    ```
    3.425 + 2/3 + 7.9
    ```

    produce el mismo resultado que la expresión

    ```
    3.425 + 7.9
    ```

    debido a que la división de números enteros de 2/3 es 0.
15. Mezclar tipos de datos en la misma expresión sin entender con claridad el efecto producido. En vista que C++ permite expresiones con tipos de datos “mixtos”, es importante entender el orden de evaluación y el tipo de datos de todos los cálculos intermedios. Como una regla general, es mejor no mezclar nunca tipos de datos en una expresión a menos que se desee un efecto específico.

    El tercero, quinto, séptimo, octavo y noveno errores en esta lista son los más comunes al inicio, mientras incluso programadores experimentados en ocasiones cometen el décimo error. Vale la pena que escriba un programa e introduzca de manera específica cada uno de estos errores, uno a la vez, para ver qué mensajes de error son producidos por el compilador, si es que se genera alguno. Entonces cuando aparezcan estos mensajes de error debido a errores inadvertidos, tendrá experiencia para comprender los mensajes y corregir los errores.

    En un nivel más fundamental, un error de programación importante cometido por los programadores principiantes es la prisa por codificar y ejecutar un programa antes que el programador haya entendido por completo lo que se requiere y los algoritmos y procedimientos que se usarán para producir el resultado deseado. Un síntoma de este apresuramiento por introducir un programa en la computadora es la carencia de un esbozo del programa propuesto o de un programa escrito en sí. Pueden evitarse muchos problemas con sólo revisar una copia del programa, ya sea manuscrita o enlistada por la computadora, antes que sea compilado.

2.9 RESUMEN DEL CAPÍTULO

1. Un programa en C++ consiste de uno o más módulos llamados funciones. Una de estas funciones debe llamarse `main()`. La función `main()` identifica el punto de inicio de un programa C++.
2. El programa C++ más simple consiste en una sola función `main()`.
3. Después del nombre de la función, el cuerpo de una función tiene la siguiente forma general:

```
{
   Todas las instrucciones de C++ van aquí;
}
```

4. Todas las instrucciones de C++ deben terminar con un punto y coma.
5. En este capítulo se introdujeron tres tipos de datos: enteros, de punto flotante y booleanos. C++ reconoce cada uno de estos tipos de datos, además de otros que aún falta por presentar.
6. El objeto `cout` puede usarse para desplegar todos los tipos de datos de C++.
7. Cuando se usa el objeto `cout` dentro de un programa, debe colocarse el comando preprocesador `#include <iostream>` en la parte superior del programa. Los comandos preprocesadores no terminan con punto y coma.
8. Toda variable en un programa C++ debe declararse como el tipo de valor que puede almacenar. Las declaraciones dentro de una función pueden colocarse en cualquier parte dentro de la función, aunque una variable sólo puede ser usada después de ser declarada. Las variables también pueden inicializarse cuando son declaradas. Además, las variables del mismo tipo pueden declararse usando una sola instrucción de declaración. Las instrucciones de declaración de variables tienen la forma general:

   tipo-de-datos nombre(s)DeVariables;

9. Un programa C++ simple que contiene instrucciones de declaración tiene la forma típica

```
#include <iostream>
using namespace std;

int main()
{
  instrucciones de declaración;

  otras instrucciones;

  return 0;
}
```

Aunque las instrucciones de declaración pueden colocarse en cualquier parte dentro del cuerpo de la función, sólo puede utilizarse una variable después que se ha declarado.

10. Las instrucciones de declaración siempre efectúan una tarea de software que consiste en informar al compilador los nombres válidos de las variables de una función. Cuando una declaración de variable también causa que la computadora asigne ubicaciones de memoria para la variable, la instrucción de declaración también se llama instrucción de definición. (Todas las declaraciones que se han usado en este capítulo han sido también instrucciones de definición.)
11. El operador `sizeof()` puede utilizarse para determinar la cantidad de almacenamiento reservado para las variables.

Consideración de opciones de carrera

Ingeniería eléctrica

La ingeniería eléctrica tiene que ver con la aplicación de los principios de la electricidad y el electromagnetismo para la fabricación de toda clase de máquinas y dispositivos que utilicen electricidad o produzcan energía eléctrica. Este campo es el más grande de todos los campos de la ingeniería. En sus inicios, a mediados del siglo XIX, sólo se interesaba por la generación de energía eléctrica. Ha evolucionado para convertirse en un campo con horizontes más amplios, abarcando las comunicaciones, las computadoras, la robótica, los dispositivos de estado sólido y el diseño de circuitos integrados.

1. Potencia. Esta área implica la generación de energía eléctrica en grandes plantas de combustibles fósiles, nucleares, solares o hidroeléctricas, o la utilización eficiente de la energía eléctrica por medio de motores o dispositivos de iluminación. También son importantes la transmisión y distribución de energía eléctrica por medio de líneas eléctricas aéreas, microondas, ductos y líneas de superconductores.
2. Electrónica de estado sólido. A través de la física moderna y la ciencia de los materiales, se han desarrollado materiales semiconductores exóticos que se usan para construir microcircuitos para la vigilancia y control de las operaciones de toda clase de dispositivos, desde juegos de video hasta robots en líneas de montaje. La mejora en la confiabilidad, la rápida reducción en el tamaño y la reducción en los requerimientos de potencia de los componentes eléctricos miniaturizados modernos han creado oportunidades ilimitadas para sus aplicaciones.
3. Comunicaciones. Las comunicaciones implican el diseño y construcción de equipo usado en la transmisión de información por medio de electricidad u ondas electromagnéticas (radio, luz, microondas, etc.). El uso del láser para la comunicación es un tema de interés contemporáneo, mientras que las características de las antenas y el radar son un poco más antiguas.
4. Computadoras y robótica. Aunque la electrónica tiene que ver con los principios asociados con las funciones de los componentes, los ingenieros en computación están interesados en diseñar los circuitos complejos que entrelazan los componentes en una computadora. Los microprocesadores, o computadoras pequeñas, están diseñados para vigilar y controlar de manera constante las operaciones de una pieza de equipo particular como un torno o un piloto automático.

CAPÍTULO 3

Asignación, formateo y entrada interactiva

TEMAS

3.1 OPERACIONES DE ASIGNACIÓN
COERCIÓN
VARIACIONES DE ASIGNACIÓN
ACUMULACIÓN
CONTEO

3.2 DAR FORMATO A NÚMEROS PARA LA SALIDA DEL PROGRAMA

3.3 EMPLEO DE LA BIBLIOTECA DE FUNCIONES MATEMÁTICAS
MOLDES

3.4 ENTRADA DE DATOS AL PROGRAMA USANDO EL OBJETO `cin`
UNA PRIMERA MIRADA A LA VALIDACIÓN DE ENTRADAS DEL USUARIO

3.5 CONSTANTES SIMBÓLICAS
COLOCACIÓN DE INSTRUCCIONES

3.6 APLICACIONES
APLICACIÓN 1: LLUVIA ÁCIDA
APLICACIÓN 2: APROXIMACIÓN A LA FUNCIÓN EXPONENCIAL

3.7 ERRORES COMUNES DE PROGRAMACIÓN

3.8 RESUMEN DEL CAPÍTULO

3.9 UN ACERCAMIENTO MÁS A FONDO: ERRORES DE PROGRAMACIÓN
CONSIDERACIÓN DE LAS OPCIONES DE CARRERA: INGENIERÍA MECÁNICA

En el capítulo anterior se exploró cómo se despliegan los resultados usando el objeto `cout` *de C++ y cómo se almacenan y se procesan los datos numéricos usando variables e instrucciones de asignación. En este capítulo se completará la introducción a C++ presentando capacidades de procesamiento y entrada adicionales.*

3.1 OPERACIONES DE ASIGNACIÓN

Ya se han encontrado instrucciones de asignación simples en el capítulo 2. Dichas instrucciones son las más básicas de C++ tanto para asignar valores a las variables como para llevar a cabo cálculos. Esta instrucción tiene la sintaxis:

variable = expresión;

La expresión más simple en C++ es una sola constante. En cada una de las siguientes instrucciones de asignación, el operando a la derecha del signo de igual es una constante:

```
largo = 25;
ancho = 17.5;
```

En cada una de estas instrucciones de asignación el valor de la constante a la derecha del signo de igual se asigna a la variable a la izquierda del signo de igual. Es importante señalar que el signo de igual en C++ no tiene el mismo significado que un signo de igual en álgebra. El signo de igual en una instrucción de asignación le indica a la computadora que determine primero el valor del operando a la derecha del signo de igual y luego almacene (o asigne) ese valor en las ubicaciones asociadas con la variable a la izquierda del signo de igual. En este sentido, la instrucción de C++ `largo = 25`; se lee "a largo se le asignó el valor 25". Los espacios en blanco en la instrucción de asignación se insertan sólo para legibilidad.

Recuerde que una variable puede ser inicializada cuando se declara. Si no se hace una inicialización dentro de la instrucción de declaración, a la variable debe asignársele un valor con una instrucción de asignación u operación de entrada antes que se use en cualquier cálculo. Pueden usarse, por supuesto, instrucciones de asignación subsiguientes para cambiar el valor asignado a una variable. Por ejemplo, suponga que las siguientes instrucciones se ejecutan una tras otra y que no se inicializó pendiente cuando fue declarada:

```
pendiente = 3.7;
pendiente = 6.28;
```

La primera instrucción de asignación le da el valor de 3.7 a la variable nombrada `pendiente`.[1] La siguiente instrucción de asignación causa que la computadora asigne un valor de 6.28 a `pendiente`. El 3.7 que estaba en `pendiente` es sobrescrito con el nuevo valor de 6.28 debido a que una variable sólo puede almacenar un valor a la vez. A veces es útil pensar en la variable a la izquierda del signo de igual como un cajón de estacionamiento temporal en un estacionamiento enorme. Del mismo modo en que un cajón de estacionamiento individual sólo puede ser usado por un automóvil a la vez, cada variable sólo puede almacenar un valor a la vez. "Estacionar" un valor nuevo en una variable causa de manera automática que el programa elimine cualquier valor estacionado ahí con anterioridad.

[1]En vista que ésta es la primera vez que se asigna un valor de manera explícita a esta variable con frecuencia se le denomina inicialización. Esto se deriva del uso histórico que expresa que una variable era inicializada la primera vez que se le asignaba un valor. Bajo este uso es correcto decir qué "pendiente fue inicializada a 3.7". Desde un punto de vista de la implementación, sin embargo, este último planteamiento es incorrecto. Esto se debe a que la operación de asignación es manejada en forma diferente por el compilador de C++ que una inicialización realizada cuando se crea una variable por una instrucción de declaración. Esta diferencia sólo es importante cuando se usan características de clase de C++ y se explica con detalle en la sección 9.1.

Además de ser una constante, el operando a la derecha del signo de igual en una instrucción de asignación puede ser una variable o cualquier otra expresión válida de C++. Una **expresión** es cualquier combinación de constantes, variables y llamadas a funciones que pueden evaluarse para producir un resultado. Por tanto, la expresión en una instrucción de asignación puede usarse para realizar cálculos usando los operadores aritméticos introducidos en la sección 2.4. Son ejemplos de instrucciones de asignación que usan expresiones que contienen estos operadores

```
suma = 3 + 7;
dif = 15 - 6;
producto = .05 * 14.6;
conteo = contador + 1;
totalnuevo = 18.3 + total;
impuestos = .06 * cantidad;
pesoTotal = factor * peso;
promedio = suma / elementos;
pendiente = (y2 - y1) / (x2 - x1);
```

Como siempre en una instrucción de asignación, el programa calcula primero el valor de la expresión a la derecha del signo de igual y luego almacena este valor en la variable a la izquierda del signo de igual. Por ejemplo, en la instrucción de asignación `pesoTotal = factor * peso;` la expresión `factor * peso` se evalúa primero para producir un resultado. Este resultado, el cual es un número, se almacena luego en la variable `pesoTotal`.

Al escribir expresiones de asignación, debe tener en cuenta dos consideraciones importantes. En vista que la expresión a la derecha del signo de igual se evalúa primero, a todas las variables usadas en la expresión debe habérseles dado valores válidos para que el resultado tenga sentido. Por ejemplo, la instrucción de asignación `pesoTotal = factor * peso;` causa que un número válido sea almacenado en `pesoTotal` sólo si el programador tiene cuidado primero de asignar números válidos a `factor` y `peso`. Por tanto la secuencia de instrucciones

```
factor = 1.06;
peso = 155.0;
pesoTotal = factor * peso;
```

nos indica los valores que se están usando para obtener el resultado que se almacenará en `pesoTotal`. La figura 3.1 ilustra los valores almacenados en las variables `factor`, `peso` y `pesoTotal`.

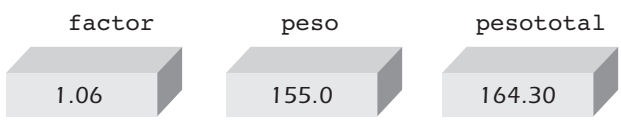


Figura 3.1 *Valores almacenados en las variables.*

La segunda consideración a tener en cuenta es que en vista que el valor de una expresión es almacenado en la variable a la izquierda del signo de igual, sólo una variable puede escribirse en esta posición. Por ejemplo, la instrucción de asignación

```
cantidad + 1892 = 1000 + 10 * 5;
```

es inválida. La expresión del lado derecho da por resultado el número entero 1050, el cual sólo puede ser almacenado en una variable. Debido a que `cantidad + 1892` no es un nombre de variable válido, el compilador no sabe dónde almacenar el valor calculado.

El programa 3.1 ilustra el uso de instrucciones de asignación para calcular el volumen de un cilindro. Como se ilustra en la figura 3.2, el volumen de un cilindro está determinado por la fórmula *volumen en* = $\pi r^2 h$, donde r es el radio del cilindro, h es la altura y π es la constante 3.1416 (con una precisión de cuatro cifras decimales).

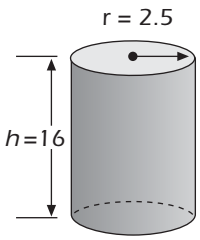


Figura 3.2 *Determinar el volumen de un cilindro.*

Programa 3.1

```
//   este programa calcula el volumen de un cilindro,
//      dados su radio y altura
#include <iostream>
using namespace std;

int main()
{
  double radio, altura, volumen;

  radio = 2.5;
  altura = 16.0;
  volumen = 3.1416 * radio * radio * altura;
  cout << "El volumen del cilindro es " << volumen << endl;

  return 0;
}
```

Cuando el programa 3.1 es compilado y ejecutado, la salida es

```
El volumen del cilindro es 314.16
```

Considere el flujo de control que usa la computadora para ejecutar el programa 3.1. La ejecución del programa comienza con la primera instrucción dentro del cuerpo de la función `main()` y continúa en forma secuencial, instrucción por instrucción, hasta que se encuentre

la llave de cierre de main. Este flujo de control se aplica a todos los programas. La computadora funciona con una instrucción a la vez, ejecutando esa instrucción sin saber cuál será la siguiente instrucción. Esto explica por qué todos los operandos usados en una expresión deben tener valores asignados a ellos antes que se evalúe la expresión. Cuando la computadora ejecuta la instrucción

```
volumen = 3.1416 * radio * radio * altura;
```

en el programa 3.1, usa cualquier valor que esté almacenado en las variables `radio` y `altura` en el momento en que se ejecuta la instrucción de asignación.[2] Si no se han asignado valores de manera específica a estas variables antes que se usen en la instrucción de asignación, la computadora usa los valores que tengan estas variables cuando se haga referencia a ellas. (En algunos sistemas todas las variables son inicializadas de manera automática en cero.) La computadora no "ve hacia delante" para verificar si se asignan valores a estas variables más adelante en el programa.

Es importante percatarse que en C++ el signo de igual, =, usado en instrucciones de asignación es en sí un operador, el cual difiere de la forma en que la mayor parte de otros lenguajes de alto nivel procesa este símbolo. En C++ (como en C), el símbolo = se llama **operador de asignación**, y una expresión que usa este operador, como `interés = principal * tasa`, es una **expresión de asignación**. En vista que el operador de asignación tiene una precedencia menor que cualquier otro operador aritmético, el valor de cualquier expresión a la derecha del signo de igual será evaluado primero, antes de la asignación.

Como todas las expresiones, las expresiones de asignación en sí mismas tienen un valor. El valor de la expresión de asignación completa es el valor asignado a la variable en el lado izquierdo del operador de asignación. Por ejemplo, la expresión `a = 5` asigna un valor de *5* a la variable a y produce que la expresión en sí tenga un valor de *5*. El valor de la expresión siempre puede verificarse usando una instrucción como

```
cout << "El valor de la expresión es " << (a = 5);
```

Aquí, el valor de la expresión en sí es desplegado y no el contenido de la variable a. Aunque tanto el contenido de la variable como la expresión tienen el mismo valor, vale la pena observar que se está tratando con dos entidades distintas.

Desde una perspectiva de programación, es la asignación real de un valor a una variable la que es significativa en una expresión de asignación; el valor final de la expresión de asignación en sí es de poca consecuencia. Sin embargo, el hecho que las expresiones de asignación tengan un valor tiene implicaciones que deben considerarse cuando se presenten los operadores relacionales en C++.

Cualquier expresión que se termine con un punto y coma se convierte en una instrucción de C++. El ejemplo más común de esto es la instrucción de asignación, la cual tan sólo es una expresión de asignación terminada con un punto y coma. Por ejemplo, terminar la expresión de asignación `a = 33` con un punto y coma produce la instrucción de asignación `a = 33;` la cual puede usarse en un programa en una sola línea.

En vista que el signo de igual es un operador en C++, son posibles en la misma expresión asignaciones múltiples o una instrucción equivalente. Por ejemplo, en la expresión `a = b = c = 25` todos los operadores de asignación tienen la misma precedencia. En vista que el ope-

[2]En vista que C++ no tiene un operador de exponenciación, el cuadrado del radio se obtiene por el término radio * radio. En la sección 3.3 se introduce la función de potencia `pow()` de C++, la cual permite elevar un número a una potencia.

rador de asignación tiene una asociatividad de derecha a izquierda, la evaluación final procede en la secuencia

```
c = 25
b = c
a = b
```

En este caso, esto tiene el efecto de asignar el número 25 a cada una de las variables en forma individual y puede representarse como

```
a = (b = (c = 25))
```

Añadir un punto y coma a la expresión original produce la instrucción de asignación múltiple

```
a = b = c = 25;
```

Esta última instrucción asigna el valor de 25 a las tres variables individuales equivalentes al siguiente orden:

```
c = 25;
b = 25;
a = 25;
```

Coerción

Algo que se debe tener en cuenta cuando se trabaje con instrucciones de asignación es el tipo de datos asignado a los valores en ambos lados de la expresión, porque ocurren conversiones de tipos de datos a lo largo de los operadores de asignación. En otras palabras, el valor de la expresión en el lado derecho del operador de asignación será convertido en el tipo de datos de la variable a la izquierda del operador de asignación. Este tipo de conversión se conoce como **coerción** porque el valor asignado a la variable en el lado izquierdo del operador de asignación es forzado al tipo de datos de la variable a la que es asignado. Un ejemplo de una coerción ocurre cuando se asigna un valor de número entero a una variable de número real; esto causa que el entero se convierta en un valor real. Por tanto, asignar un valor entero a una variable real causa que el entero sea convertido en un valor real. Del mismo modo, asignar un valor real a una variable entera fuerza la conversión de un valor real a un entero, lo cual produce la pérdida de la parte fraccionaria del número debido a truncamiento. Por ejemplo, si `temp` es una variable entera, la asignación `temp = 25.89` causa que el valor entero 25 se almacene en la variable entera `temp`.[3]

Un ejemplo más completo de conversiones de tipos de datos, las cuales incluyen conversión en modo mixto y conversión de asignación, es la evaluación de la expresión

```
a = b * d
```

[3]Es evidente que la porción entera correcta sólo se conserva cuando está dentro del rango de enteros permitidos por el compilador.

Punto de Información

lvalues y rvalues

Los términos `lvalue` y `rvalue` se utilizan con frecuencia en la tecnología de programación. Ambos términos son independientes del lenguaje y significan lo siguiente: un **`lvalue`** puede tener un valor asignado mientras que un **`rvalue`** no puede tenerlo.

Tanto en C como en C++ esto significa que un `lvalue` puede aparecer tanto en el lado izquierdo como en el derecho de un operador de asignación mientras que un `rvalue` sólo puede aparecer en el lado derecho de un operador de asignación. Por ejemplo, cada variable que hemos encontrado puede ser un `lvalue` o un `rvalue`, mientras que un número sólo puede ser un `rvalue`. Sin embargo, no todas las variables pueden ser `lvalue` y `rvalue`. Por ejemplo, un tipo de arreglo, el cual se introduce en el capítulo 11, no puede ser un `lvalue` o un `rvalue`, mientras que los elementos individuales del arreglo pueden ser ambos.

donde `a` y `b` son variables enteras y `d` es una variable de precisión simple. Cuando se evalúa la expresión en modo mixto `b * d`,[4] el valor de `d` usado en la expresión es convertido a un número de precisión doble para propósitos de cálculo. (Es importante señalar que el valor almacenado en `d` se mantiene como un número de precisión simple.) En vista que uno de los operandos es una variable de precisión doble, el valor de la variable entera `b` es convertido en un número de precisión doble para el cálculo (una vez más, el valor almacenado en `b` sigue siendo un entero) y el valor resultante de la expresión `b * d` es un número de precisión doble. Por último, se aplica la conversión del tipo de datos por medio del operador de asignación. En vista que el lado izquierdo del operador de asignación es una variable entera, el valor de precisión doble de la expresión (`b * d`) será truncado a un valor entero y almacenado en la variable `a`.

Variaciones de asignación

Auque sólo se permite una variable inmediatamente a la izquierda del signo de igual en una expresión de asignación, la variable a la izquierda del signo de igual también puede usarse a la derecha del signo de igual. Por ejemplo, la expresión de asignación `suma = suma + 10` es válida. Es claro que, como una expresión algebraica, `suma` nunca podría ser igual a sí misma más 10. Pero en C++, la expresión `suma = suma + 10` no es una ecuación, es una expresión que se evalúa en dos pasos importantes. El primer paso es calcular el valor de `suma + 10`. El segundo paso es almacenar el valor calculado en `suma`. Vea si puede determinar la salida del programa 3.2.

[4]Si es necesario, repase las reglas en la sección 2.4 para la evaluación de expresiones de modo mixto.

Programa 3.2

```
#include <iostream>
using namespace std;

int main()
{
  int suma;

  suma = 25;
  cout << "El número almacenado en suma es " << suma << endl;
  suma = suma + 10;
  cout << "El número almacenado ahora en suma es " << suma << endl;

  return 0;
}
```

La instrucción de asignación `suma = 25;` le indica a la computadora que almacene el número 25 en suma, como se muestra en la figura 3.3.

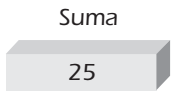


Figura 3.3 *El entero 25 es almacenado en* `suma`.

La primera instrucción `cout` causa que el valor almacenado en suma sea desplegado por el mensaje `El número almacenado en suma es 25`. La segunda instrucción de asignación en el programa 3.2, `suma = suma + 10;` causa que el programa recupere el 25 almacenado en `suma` y agregue 10 a este número, produciendo el número 35. Entonces el número 35 se almacena en la variable en el lado izquierdo del signo de igual, la cual es la variable `suma`. El 25 que estaba en suma tan sólo se sobrescribe con el nuevo valor de 35, como se muestra en la figura 3.4.

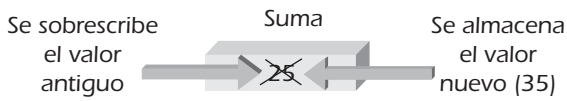


Figura 3.4 `suma = suma + 10;` *causa que se almacene un nuevo valor en* `suma`.

Las expresiones de asignación como `suma = suma + 25`, que usan la misma variable en ambos lados del operador de asignación, pueden escribirse usando los siguientes atajos de **operadores de asignación**:

```
+=   -=   *=   /=   %=
```

Por ejemplo, la expresión `suma = suma + 10` puede escribirse como `suma += 10`. Del mismo modo, la expresión `precio *= tasa` es equivalente a la expresión `precio = precio * tasa`.

Al utilizar estos operadores de asignación es importante observar que la variable de la izquierda del operador de asignación se aplica a la expresión de la derecha completa. Por ejemplo `precio * = tasa + 1` es equivalente a la expresión `precio = precio * (tasa + 1)`, no `precio = precio * tasa + 1.`

Acumulación

Las expresiones de asignación como `suma += 10` o su equivalente, `suma = suma + 10`, son muy comunes en programación. Estas expresiones se requieren para acumular subtotales cuando los datos se introducen un número a la vez. Por ejemplo, si se desea sumar los números 96, 70, 85 y 60 en forma de calculadora, podrían usarse las siguientes instrucciones:

Instrucción	Valor en `suma`
`suma = 0;`	0
`suma = suma + 96;`	96
`suma = suma + 70;`	166
`suma = suma + 85;`	251
`suma = suma + 60;`	311

La primera instrucción inicializa `suma en 0`. Esto elimina cualquier número ("valor inservible") almacenado en `suma` que pudiera invalidar el total final. Conforme se agrega cada número, el valor almacenado en `suma` se incrementa en forma correspondiente. Después de completar la última instrucción, `suma` contiene el total de todos los números agregados. El programa 3.3 ilustra el efecto de estas instrucciones al desplegar el contenido de `suma` después de hacer cada adición.

Programa 3.3

```
#include <iostream>
using namespace std;

int main()
{
  int suma;

  suma = 0;
  cout << "El valor de suma se estableció en forma inicial en " << suma << endl;
  suma = suma + 96;
  cout << "  suma ahora es " << suma << endl;
  suma = suma + 70;
  cout << "  suma ahora es " << suma << endl;
  suma = suma + 85;
  cout << "  suma ahora es " << suma  << endl;
  suma = suma + 60;
  cout << "  La suma final es " << suma << endl;

  return 0;
}
```

La salida desplegada por el programa 3.3 es:

```
El valor de suma se estableció en forma inicial en 0
  suma ahora es 96
  suma ahora es 166
  suma ahora es 251
  La suma final es 311
```

Aunque el programa 3.3 no es un programa práctico (es más fácil sumar los números en forma manual), ilustra el efecto subtotalizador del uso repetido de instrucciones que tienen la forma

variable = variable + Valornuevo;

Se encontrarán muchos usos para este tipo de **instrucción de acumulación** cuando nos familiaricemos más con las instrucciones de repetición introducidas en el capítulo 5.

Conteo

Una instrucción de asignación que es muy similar a la instrucción de acumulación es la instrucción de conteo. Las instrucciones de conteo tienen la forma

variable = variable + númeroFijo;

Son ejemplos de instrucciones de conteo

```
i = i + 1;
n = n + 1;
contador = contador + 1;
j = j + 2;
m = m + 2;
kk = kk + 3;
```

En cada uno de estos ejemplos se usa la misma variable en ambos lados del signo de igual. Después de ejecutar la instrucción, el valor de la variable respectiva se incrementa en una cantidad fija. En los primeros tres ejemplos las variables `i, n` y `contador` fueron incrementadas en uno. En los siguientes dos ejemplos las variables respectivas se han incrementado en dos y en el ejemplo final la variable `kk` se ha incrementado en tres.

Para el caso especial en que una variable es incrementada o disminuida en uno, C++ proporciona dos operadores unitario. Usando el operador de incremento,[5] `++`, la expresión `variable = variable + 1` puede ser reemplazada por la expresión `variable++` o `++variable`. Son ejemplos del operador de incremento

Expresión	Alternativa
`i = i + 1`	`i++ o ++i`
`n = n + 1`	`n++ o ++n`
`contador = contador + 1`	`contador++ o ++contador`

El programa 3.4 ilustra el uso del operador de incremento.

[5]Como una nota histórica, el ++ en C++ se inspiró en el símbolo del operador de incremento. Se usó para indicar que C++ fue el siguiente incremento en el lenguaje C.

Programa 3.4

```
#include <iostream>
using namespace std;

int main()
{
  int contador;

  contador = 0;
  cout << "El valor inicial de contador es " << contador << endl;
  contador++;
  cout << "   contador es ahora " << contador << endl;
  contador++;
  cout << "   contador es ahora " << contador << endl;
  contador++;
  cout << "   contador es ahora " << contador << endl;
  contador++;
  cout << "   contador es ahora " << contador << endl;

  return 0;
}
```

La salida desplegada por el programa 3.4 es:

```
El valor inicial de contador es 0
   contador es ahora 1
   contador es ahora 2
   contador es ahora 3
   contador es ahora 4
```

Cuando el operador `++` aparece antes de una variable se llama **operador de prefijo para incremento**; cuando aparece después de una variable se llama **operador de postfijo para incremento**. La distinción entre un operador de prefijo de incremento y uno de postfijo es importante cuando la variable que es incrementada se usa en una expresión de asignación. Por ejemplo, la expresión `k = ++n` hace dos cosas en una expresión. Al principio, el valor de `n` es incrementado en uno y luego el valor nuevo de `n` es asignado a la variable `k`. Por tanto, la instrucción `k = ++n;` es equivalente a las dos instrucciones

```
n = n + 1;   // incrementa n primero
k = n;       // asigna el valor de n a
```

La expresión de asignación `k = n++`, la cual usa un operador de sufijo para incremento, invierte este procedimiento. Un postfijo para incremento opera después que se ha comple-

tado la asignación. Por tanto, la instrucción `k = n++;` asigna primero el valor actual de `n` a `k` y luego incrementa el valor de `n` en uno. Esto es equivalente a las dos instrucciones

```
k = n;          // asigna el valor de n a k
n = n + 1;      // y luego incrementa n
```

Además del operador de incremento, C++ también proporciona un operador de decremento, `--`. Como podría esperarse, las expresiones `variable--` y `--variable` son equivalentes a la expresión `variable = variable - 1`.

Son ejemplos del operador de decremento:

Expresión	Alternativa
`i = i - 1`	`i-- o --i`
`n = n - 1`	`n-- o --n`
`contador = contador - 1`	`contador-- o --contador`

Cuando el operador `--` aparece antes de una variable se llama **operador de prefijo para decremento**. Cuando el decremento aparece después de una variable se llama **operador de postfijo para decremento**. Por ejemplo, las expresiones `n--` y `--n` reducen el valor de `n` en uno. Estas expresiones son equivalentes a la expresión más larga `n = n - 1`. Sin embargo, como con los operadores de incremento, los operadores de prefijo y postfijo para decremento producen resultados diferentes cuando se usan en expresiones de asignación. Por ejemplo, la expresión `k = --n` disminuye primero el valor de `n` en uno antes de asignar el valor de `n` a `k`, mientras la expresión `k = n--` signa primero el valor actual de `n` a `k` y luego reduce el valor de `n` en uno.

Ejercicios 3.1

1. Escriba una instrucción de asignación para calcular la circunferencia de un círculo que tiene un radio de 3.3 pulgadas. La ecuación para determinar la circunferencia, *c,* de un círculo es $c = 2\pi r$, donde r es el radio y π es igual a 3.1416.
2. Escriba una instrucción de asignación para calcular el área de un círculo. La ecuación para determinar el área, *a*, de un círculo es $a = \pi r^2$, donde r es el radio y $\pi = 3.1416$.
3. Escriba una instrucción de asignación para convertir temperatura en grados Fahrenheit a grados Celsius. La ecuación para esta conversión es Celsius = 5/9 (Fahrenheit – 32).
4. Escriba una instrucción de asignación para calcular la distancia de un viaje redondo, *d*, en pies, de un viaje de *s* millas en un solo sentido.
5. Escriba una instrucción de asignación para calcular el tiempo transcurrido, en minutos, necesario para hacer un viaje. La ecuación para calcular el tiempo transcurrido es *tiempo = distancia total / velocidad promedio*. Suponga que la distancia debe ser en millas y la velocidad promedio en millas/hora.

6. Escriba una instrucción de asignación para calcular el *enésimo* término en una secuencia aritmética. La fórmula para calcular el valor, v, del *enésimo* término es $v = a + (n - 1)d$, donde a = el primer número en la secuencia y d = la diferencia entre cualesquiera dos números en la secuencia.

7. Escriba una instrucción de asignación para calcular la expansión lineal en una viga de acero como una función del aumento de temperatura. La fórmula para la expansión lineal, l, es $l = l_o[1+\propto(T_f - T_o)]$, donde l_o es el largo de la viga a la temperatura T_o, $\propto$ es el coeficiente de expansión lineal y T_f es la temperatura final de la viga.

8. La ley de Coulomb establece que la fuerza F, actuando entre dos esferas cargadas eléctricamente, está dada por la fórmula $F = kq_1q_2/r^2$, donde q_1 es la carga en la primera esfera, q_2 es la carga en la segunda esfera, r es la distancia entre los centros de las dos esferas y k es una constante de proporcionalidad. Escriba una instrucción de asignación para calcular la fuerza, F.

9. Escriba una instrucción de asignación para determinar el momento de flexión máximo, M, de una viga. La fórmula para el momento de flexión máximo es, $M = XW(L - X) / L$, donde X es la distancia desde el extremo de la viga donde se aplica un peso, W, y L es el largo de la viga.

10. Determine la salida del siguiente programa:

```
#include <iostream>
using namespace std;

int main() // un programa que ilustra el truncamiento de enteros
{
   int num1, num2;

   num1 = 9/2;
   num2 = 17/4;
   cout << "el primer entero desplegado es " << num1 << endl;
   cout << "el segundo entero desplegado es " << num2 << endl;

  return 0;
}
```

11. Determine y corrija los errores en los siguientes programas.

    a.
```
#include <iostream>
  using namespace std;
  int main()
  {
    ancho = 15
    area = largo * ancho;
    cout << "El area es " << area

  }
```

b.
```
#include <iostream>
using namespace std;
int main()
{
  int largo, ancho, area;
  area = largo * ancho;
  largo = 20;
  ancho = 15;
  cout << "El area es " << area;

  return 0;
```

c.
```
#include <iostream.h>

 int main()
 {
   int largo = 20; ancho = 15, area;
   largo * ancho = area;
   cout << "El area es " , area;

   return 0;
 }
```

12. Por error un estudiante reordenó las instrucciones en el programa 3.3 como sigue:

```
#include <iostream>
using namespace std;

int main()
{
  int suma;
  suma = 0;
  suma = suma + 96;
  suma = suma + 70;
  suma = suma + 85;
  suma = suma + 60;
  cout << "El valor de suma se estableció en forma inicial en "
          << suma << endl;
  cout << "  suma ahora es " << suma << endl;
  cout << "  suma ahora es " << suma << endl;
  cout << "  suma ahora es " << suma << endl;
  cout << "  La suma final es " << suma << endl;

  return 0;
}
```

Determine la salida que produce este programa.

13. Usando el programa 3.1, determine el volumen de cilindros que tienen los siguientes radios y alturas.

Radio (pulg.)	Altura (pulg.)
1.62	6.23
2.86	7.52
4.26	8.95
8.52	10.86
12.29	15.35

14. El área de una elipse (véase la figura 3.5) está dada por la fórmula Área = πab.

Usando esta fórmula, escriba un programa C++ para calcular el área de una elipse que tenga un eje menor, *a*, de 2.5 pulgadas y un eje mayor, *b*, de 6.4 pulgadas.

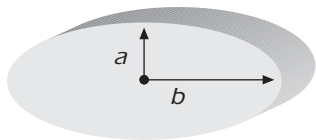


Figure 3.5 El eje menor *a* y el eje mayor *b* de una elipse.

15. Modifique el programa 3.1 para calcular el peso, en libras, del cilindro de acero cuyo volumen fue encontrado por el programa. Para determinar el peso la fórmula es peso = 0.28 $(\pi)(r^2)(h)$, donde *r* es el radio (en pulgadas) y *h* es la altura (en pulgadas) del cilindro.

16. La circunferencia de una elipse (véase la figura 3.5) está dada por la fórmula:

$$\text{Circunferencia} = \pi\sqrt{(a+b)^2}$$

Usando esta fórmula, escriba un programa en C++ para calcular la circunferencia de una elipse que tiene un radio menor de 2.5 pulgadas y un radio mayor de 6.4 pulgadas. (*Sugerencia:* la raíz cuadrada puede obtenerse elevando la cantidad $2[a^2 + b^2]$ a la potencia 0.5.)

17. a. La resistencia combinada de tres resistores conectados en paralelo, como se muestra en la figura 3.6, está dada por la ecuación

$$\text{Resistencia combinada} = \frac{1}{\frac{1}{R_1}+\frac{1}{R_2}+\frac{1}{R_3}}$$

Escriba un programa en C++ para calcular y desplegar la resistencia combinada cuando los tres resistores $R_1 = 1000$, $R_2 = 1000$, y $R_3 = 1000$ están conectados en paralelo. Su programa deberá producir el despliegue "La resistencia combinada, en ohmios, es xxxxx", donde las x son reemplazadas por el valor de la resistencia combinada calculada por su programa.

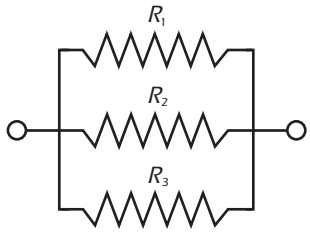


Figura 3.6 Tres resistores conectados en paralelo.

b. ¿Cómo sabe que el valor calculado por su programa es correcto?
c. Una vez que ha verificado la salida producida por su programa, modifíquela para determinar la resistencia combinada cuando los resistores $R_1 = 1500$, $R_2 = 1200$, y $R_3 = 2000$ están conectados en paralelo.

18. a. Escriba un programa en C++ para calcular y desplegar el valor de la pendiente de la línea que conecta dos puntos cuyas coordenadas son (3, 7) y (8, 12). Use el hecho que la pendiente entre dos puntos que tienen coordenadas (x_1, y_1) y (x_2, y_2) es pendiente = $(y_2 - y_1) / (x_2 - x_1)$. Su programa deberá producir el despliegue "La pendiente es xxxx", donde las x son reemplazadas por el valor calculado por su programa.
b. ¿Cómo sabe que el resultado producido por su programa es correcto?
c. Una vez que ha verificado la salida producida por su programa, modifíquela para determinar la pendiente de la línea que conecta los puntos (2, 10) y (12, 6).

19. a. Escriba un programa en C++ para calcular y desplegar las coordenadas del punto medio de la línea que conecta los dos puntos dados en el ejercicio 18a. Use el hecho que las coordenadas del punto medio entre dos puntos que tienen coordenadas (x_1,y_1) y (x_2,y_2) son $[(x_1 + x_2)]/2$, $[(y_1 + y_2)]/2)$. Su programa deberá producir el siguiente despliegue:

```
La coordenada x del punto medio es xxx
La coordenada y del punto medio es xxx
```

donde las x son reemplazadas con los valores calculados por su programa.
b. ¿Cómo sabe que los valores del punto medio calculados por su programa son correctos?
c. Una vez que ha verificado la salida producida por su programa, modifíquelo para determinar las coordenadas del punto medio de la línea que conecta los puntos (2, 10) y (12, 6).

20. a. Para el circuito eléctrico mostrado en la figura 3.7, las corrientes en los ramales, i_1, i_2, e i_3 pueden determinarse usando las fórmulas

$$i_1 = \frac{E_2R_3 + E_1(R_1 + R_3)}{(R_1 + R_3)(R_2 + R_3) - (R_3)^2}$$

$$i_2 = \frac{E_1R_3 + E_2(R_1 + R_3)}{(R_1 + R_3)(R_2 + R_3) - (R_3)^2}$$

$$i_3 = i_1 - i_2$$

Usando estas fórmulas escriba un programa en C++ para calcular las corrientes en los ramales cuando R_1 = 10 ohmios, R_2 = 4 ohmios, R_3 = 6 ohmios, E_1 = 12 voltios y E_2 = 9 voltios. El despliegue producido por su programa deberá ser

```
La corriente en el ramal 1 es xxxx
La corriente en el ramal 2 es xxxx
La corriente en el ramal 3 es xxxx
```

donde las x son reemplazadas por los valores determinados en su programa.

b. ¿Cómo sabe que las corrientes en circuito calculadas por su programa son correctas?

c. Una vez que ha verificado la salida producida por su programa, modifíquela para determinar las corrientes en los ramales para los siguientes valores: R_1 = 1500, R_2 = 1200, R_3 = 2000, E_1 = 15 y E_2 = 12.

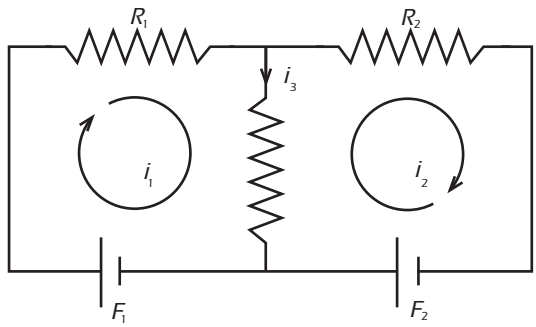


Figura 3.7 Un circuito eléctrico.

3.2 DAR FORMATO A NÚMEROS PARA LA SALIDA DEL PROGRAMA

Además de desplegar resultados correctos, es importante en extremo que un programa presente sus resultados en forma atractiva. La mayor parte de los programas son juzgados por la facilidad de introducción de datos percibida y el estilo y presentación de su salida. Por ejemplo, desplegar un resultado monetario como 1.897 no cumple con las convenciones aceptadas para los informes. El despliegue deberá ser $1.90 o $1.89, dependiendo si se usa redondeo o truncamiento.

El formato de los números desplegados por `cout` puede controlarse por manipuladores de ancho de campo incluidos en cada flujo de salida. La tabla 3.1 enumera los manipuladores disponibles que más se usan para este propósito.[6]

[6]Como se señaló en el capítulo 2, el manipulador `endl` inserta una `nueva línea` y luego vacía el flujo.

Tabla 3.1 Manipuladores de flujo más comunes

Manipulador	Acción
`setw(n)`	Establece el ancho de campo en *n*.
`setprecision(n)`	Establece la precisión del punto flotante en *n* lugares. Si se designa el manipulador `fixed`, *n* especifica el número total de dígitos desplegados después del punto decimal; de otra manera, *n* especifica el número total de dígitos significativos desplegados (números enteros más dígitos fraccionarios).
`setfill('x')`	Establece el carácter de relleno a la izquierda por omisión en *x*. (El carácter de relleno principal por omisión es un espacio, el cual es la salida para rellenar el frente de un campo de salida siempre que el ancho del campo es mayor que el valor que se está desplegando.)
`setiosflags(flags)`	Establece el formato de los indicadores (véase la tabla 3.3 para las configuraciones de los indicadores).
`scientific`	Establece la salida para desplegar números reales en notación científica.
`showbase`	Despliega la base usada para los números. Se despliega un 0 a la izquierda para los números octales y un 0x a la izquierda para números hexadecimales.
`showpoint`	Siempre despliega seis dígitos en total (combinación de partes enteras y fraccionarias). Rellena con ceros a la derecha si es necesario. Para valores enteros mayores, revierte a notación científica.
`showpos`	Despliega todos los números positivos con un signo de + a la izquierda.
`boolalpha`	Despliega valores booleanos como verdadero y falso, en lugar de como 1 y 0.
`dec`	Establece la salida para un despliegue decimal por omisión.
`endl`	Da salida a un carácter de línea nueva y despliega todos los caracteres en el búfer.
`fixed`	Siempre muestra un punto decimal y usa seis dígitos por omisión después del punto decimal. Rellena con ceros a la derecha si es necesario.
`flush`	Despliega todos los caracteres en el búfer.
`left`	Justifica a la izquierda todos los números.
`hex`	Establece la salida para un despliegue hexadecimal.
`oct`	Establece la salida para un despliegue octal.
`uppercase`	Despliega dígitos hexadecimales y el exponente en notación científica en mayúsculas.
`right`	Justifica a la derecha todos los números (éste es el valor por omisión).
`noboolalpha`	Despliega valores booleanos como 1 y 0, en lugar de verdadero y falso.
`noshowbase`	No despliega números octales con un 0 a la izquierda y los números hexadecimales con un 0x a la izquierda.
`noshowpoint`	No usa un punto decimal para números reales sin partes fraccionarias, no despliega ceros a la derecha en la parte fraccionaria de un número y despliega un máximo de sólo seis dígitos decimales.
`noshowpos`	No despliega signos de + a la izquierda (éste es el valor por omisión).
`nouppercase`	Despliega dígitos hexadecimales y el exponente en notación científica en minúsculas.

Por ejemplo, la instrucción `cout << "La suma de 6 y 5 es" << setw(3) << 21;` crea esta impresión:

```
La suma de 6 y 5 es 21
```

El manipulador de ancho de campo `setw(3)` incluido en el flujo de datos pasado a `cout` se usa para establecer el ancho del campo desplegado. El 3 en este manipulador establece el ancho de campo por omisión para el siguiente número en el flujo para que tenga un ancho de tres espacios. Esta configuración del ancho de campo causa que 21 se imprima en un campo de tres espacios, el cual incluye un espacio en blanco y el número 21. Como se ilustra, los enteros están justificados a la derecha dentro del campo especificado.

Los manipuladores de ancho de campo son útiles para imprimir columnas de números de modo que los números en cada columna se alineen en forma correcta. Por ejemplo, el programa 3.5 ilustra cómo se alinearía una columna de números enteros en ausencia de manipuladores de ancho de campo.

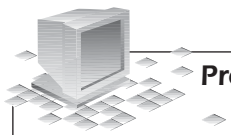
Programa 3.5

```
#include <iostream>
using namespace std;

int main()
{
  cout << 6 << endl
       << 18 << endl
       << 124 << endl
       << "---\n"
       << (6+18+124) << endl;

  return 0;
}
```

La salida del programa 3.5 es la siguiente:

```
6
18
124
---
148
```

En vista que no se incluyeron manipuladores de ancho de campo en el programa 3.5, el objeto `cout` asigna suficiente espacio para cada número conforme lo recibe. Para forzar a los números a alinearse con el dígito de las unidades se requiere un ancho de campo suficiente para el número más grande desplegado. Para el programa 3.5, un ancho de tres bastaría. El uso de este ancho de campo se ilustra en el programa 3.6.



Programa 3.6

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << setw(3) << 6 << endl
       << setw(3) << 18 << endl
       << setw(3) << 124 << endl
       << "---\n"
       << (6+18+124) << endl;

  return 0;
}
```

La salida del programa 3.6 es

```
  6
 18
124
---
148
```

El manipulador de ancho de campo debe incluirse para cada ocurrencia de un número insertado en el flujo de datos enviado a `cout`; este manipulador particular sólo se aplica a la siguiente inserción de datos inmediata. Los otros manipuladores permanecen en efecto hasta que se cambian.

Cuando se usa un manipulador que requiere un argumento debe incluirse el archivo de encabezado `iomanip` como parte del programa. Esto se logra con el comando preprocesador `#include <iomanip>`, el cual se enlista como la segunda línea en el programa 3.6.

Dar formato completo a números de punto flotante requiere el uso de tres manipuladores de ancho de campo. El primer manipulador establece el ancho total del despliegue, el segundo fuerza el despliegue de un punto decimal y el tercer manipulador determina cuántos dígitos significativos se desplegarán a la derecha del punto decimal. Por ejemplo, analice la siguiente instrucción:

```
cout << "|" << setw(10) << fixed << setprecision(3) << 25.67 << "|";
```

Causa la siguiente impresión:

```
|     25.670|
```

El símbolo de barra, |, en el ejemplo se usa para delimitar (marcar) el principio y el fin del campo de despliegue. El manipulador `setw` le indica a `cout` que despliegue el número en un campo total de 10, el manipulador `fixed` fuerza de manera explícita el despliegue de un punto decimal y designa que el manipulador `setprecision` se usa para designar el número de dígitos que se va a desplegar después del punto decimal. En este caso, `setprecision` especifica un despliegue de tres dígitos después del punto decimal. Sin la designación explícita de un punto decimal (el cual también puede designarse como setiosflags(ios::fixed)), el manipulador `setprecision` especifica el número total de dígitos desplegados, el cual incluye las partes enteras y fraccionarias del número.

Para todos los números (enteros, de precisión simple y de precisión doble), `cout` ignora la especificación del manipulador `setw` si el campo especificado total es demasiado pequeño, y asigna suficiente espacio para la parte entera del número que se va a imprimir. La parte fraccionaria de los números de precisión simple y de precisión doble es desplegada hasta la precisión establecida con el manipulador `setprecision`. (En ausencia de un manipulador `setprecision`, la precisión por omisión se establece en seis lugares decimales.) Si la parte fraccionaria del número que se va a desplegar contiene más dígitos de los indicados en el manipulador `setprecision`, el número se redondea al número indicado de lugares decimales; si la parte fraccionaria contiene menos dígitos que los especificados, el número es desplegado con menos dígitos. La tabla 3.2 ilustra el efecto de varias combinaciones de manipuladores de formato. Una vez más, por claridad, se usa el símbolo de barra, |, para delinear el principio y el fin de los campos de salida.

Tabla 3.2 Efecto de los manipuladores de formato

Manipuladores	Número	Despliegue	Comentarios
`setw(2)`	`3`	`| 3|`	El número cabe en el campo
`setw(2)`	`43`	`|43|`	El número cabe en el campo
`setw(2)`	`143`	`|143|`	El ancho de campo se ignora
`setw(2)`	`2.3`	`|2.3|`	El ancho de campo se ignora

`setw(5)`
`fixed`
`setprecision(2)`
	`2.366`	`| 2.37|`	Ancho de campo de cinco con dos dígitos decimales

`setw(5)`
`fixed`
`setprecision(2)`
	`42.3`	`|42.30|`	El número cabe en el campo con precisión especificada

`setw(5)` `setprecision(2)`	`142.364`	`|1.4e+002|`	El ancho de campo se ignora y se usa notación científica con el manipulador `setprecision` especificando el número total de dígitos significativos (enteros más fraccionarios)

Punto de Información

¿Qué es un indicador o bandera?

En la programación actual, el término indicador o **bandera** se refiere a un elemento, como una variable o argumento, que establece una condición por lo general considerada activa o inactiva. Aunque se desconoce el origen exacto de este término en programación, es probable que se haya originado del uso de banderas reales para señalar una condición, como las banderas de alto, siga, precaución y ganador que se usan por lo común en las carreras de automóviles.

De manera similar, cada argumento de indicador para la función del manipulador `setiosflags()` activa una condición específica. Por ejemplo, el indicador `ios::dec` establece el formato de despliegue decimal, y el indicador `ios::oct` activa el formato de despliegue octal. En vista que estas condiciones son mutuamente excluyentes (sólo una condición puede estar activa a la vez), activar uno de estos indicadores desactiva de manera automática los otros indicadores.

Los indicadores que no son mutuamente excluyentes, como `ios::dec`, `ios::showpoint` e `ios::fixed` pueden establecerse como activas de manera simultánea. Esto puede hacerse usando tres llamadas `setiosflag()` individuales o combinando todos los argumentos en una llamada como sigue:

```
cout << setiosflags ios::dec | ios::fixed | ios::showpoint);
```

Tabla 3.2 Efecto de los manipuladores de formato (continuación)

Manipuladores	Número	Despliegue	Comentarios

`setw(5)`
`fixed`
`setprecision(2)`
	`142.364`	`|142.36|`	Se ignora el ancho de campo pero se usa la especificación de precisión. Aquí el manipulador `setprecision` especifica el número de dígitos fraccionarios

`setw(5)`
`fixed`
`setprecision(2)`
	`142.366`	`|142.37|`	Se ignora el ancho de campo pero se usa la especificación de precisión. Aquí el manipulador `setprecision` especifica el número de dígitos fraccionarios. (Nótese el redondeo del último dígito decimal)

`setw(5)`
`fixed`
`setprecision(2)`
	`142`	`|  142|`	Se usa el ancho de campo, los manipuladores fixed y `setprecision` irrelevantes, porque el número es un entero.

Además de los manipuladores `setw` y `setprecision`, también está disponible un manipulador de justificación de campo. Como se ha visto, los números enviados a `cout` se despliegan por lo normal con justificación a la derecha en el campo de despliegue, mientras las cadenas se despliegan justificadas a la izquierda. Para alterar la justificación por omisión para un flujo de datos, puede usarse el manipulador `setiosflags`. Por ejemplo, analice la siguiente instrucción:

```
cout << "|" << setw(10) << setiosflags(ios::left) << 142 << "|";
```

Esto causa el siguiente despliegue justificado a la izquierda:

```
|142        |
```

Como se ha visto, ya que los datos pasados a `cout` pueden continuarse a lo largo de múltiples líneas, el despliegue anterior también sería producido por la instrucción:

```
cout << "|" << setw(10)
        << setiosflags(ios::left)
        << 142 << "|";
```

Como siempre, el manipulador de ancho de campo sólo está para el siguiente conjunto sencillo de datos desplegado por `cout`. La justificación a la derecha para las cadenas en un flujo se obtiene con el manipulador `setiosflags(ios::right)`. El símbolo `ios` tanto en el nombre de la función como en el argumento `ios::right` se deriva de las primeras letras de las palabras "input output stream" (flujo de entrada y salida).

Además de los indicadores de izquierda y derecha que pueden usarse con el manipulador `setiosflags()`, pueden usarse otros indicadores para afectar la salida. Los indicadores más usados para este manipulador se enumeran en la tabla 3.3. Los indicadores en esta tabla proporcionan de manera efectiva una forma alternativa para establecer los manipuladores equivalentes enumerados en la tabla 3.1.

Tabla 3.3 Indicadores de formato para usar con `setiosflags()`

Indicador	Significado
`ios::fixed`	Siempre muestra el punto decimal con seis dígitos después del punto decimal. Rellena con ceros a la derecha si es necesario. Este indicador tiene precedencia si se establece con el indicador `ios::showpoint`.
`ios::scientific`	Usa despliegue exponencial en la salida.
`ios::showpoint`	Siempre despliega un punto decimal y seis dígitos significativos en total (combinación de partes enteras y fraccionarias). Rellena con ceros a la derecha después del punto decimal si es necesario. Para valores enteros más grandes, revierte a notación científica a menos que esté establecido el indicador `ios::fixed`.
`ios::showpos`	Despliega un signo + a la izquierda cuando el número es positivo.
`ios::left`	Justifica a la izquierda la salida.
`ios::right`	Justifica a la derecha la salida.

Punto de Información

Dar formato a los datos en el flujo de `cout`

Los datos de punto flotante en un flujo de salida `cout` pueden formatearse en formas precisas. Uno de los requerimientos de formato más comunes es desplegar números en un formato monetario con dos dígitos después del punto decimal, como 123.45. Esto puede hacerse con la siguiente instrucción:

```
cout << setiosflags(ios::fixed)
     << setiosflags(ios::showpoint)
     << setprecision(12);
```

El primer indicador en el manipulador, `ios::fixed`, hace que todos los números de punto flotante colocados en el flujo `cout` se desplieguen en notación decimal. Este indicador también impide el uso de notación científica. El siguiente indicador, `ios::showpoint`, le indica al flujo que siempre despliegue un punto decimal. Por último, el manipulador `setprecision` le indica al flujo que siempre despliegue dos valores decimales después del punto decimal. En lugar de usar manipuladores, también puede usar los métodos de flujo de `cout setf()` y `precision()`. Por ejemplo, el formato anterior puede lograrse también usando el código:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

Nótese la sintaxis aquí: el nombre del objeto, `cout`, se separa del método con un punto. Ésta es la forma estándar de especificar un método y conectarlo con un objeto específico. El estilo que seleccione es cuestión de preferencia.

Además, los indicadores usados tanto en el método `setf()` como en el manipulador `setiosflags()` pueden combinarse usando el operador a nivel de bit Or, | (que se explica en la sección 15.2). Usando este operador, las siguientes dos instrucciones son equivalentes.

```
cout << setiosflags(ios::fixed | ios::showpoint);
cout.setf(ios::fixed | ios::showpoint);
```

El estilo que seleccione es cuestión de preferencia.

Debido a que los indicadores en la tabla 3.3 se usan como argumentos para el método del manipulador `setiosflags()` y debido a que los términos "argumento" y "parámetro" son sinónimos, otro nombre para un método manipulador que use argumentos es el de **manipulador parametrizado**. El siguiente es un ejemplo de métodos de manipulador parametrizado:

```
cout << setiosflags(ios::showpoint) << setprecision(4);
```

Esto hace que todos los números de punto flotante subsiguientes sean enviados al flujo de salida para ser desplegados con un punto decimal y cuatro dígitos decimales. Si el número tiene menos de cuatro dígitos decimales, se llenará con ceros a la derecha.

Además de dar salida a los enteros en notación decimal, los manipuladores `oct` y `hex` permiten conversiones a formato octal y hexadecimal, respectivamente. El programa 3.7 ilustra el uso de estos indicadores. Debido a que el formato decimal es el despliegue por omisión, el manipulador `dec` no se requiere en el primer flujo de salida.

Programa 3.7

```
// un programa que ilustra conversiones de salida
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << "El valor decimal (base 10) de 15 es " << 15 << endl;
  cout << "El valor octal (base 8) de 15 es "
       << showbase << oct << 15 <<endl;
  cout << "El valor hexadecimal (base 16) de 15 es "
         << showbase << hex << 15 << endl;

  return 0;
}
```

La salida producida por el programa 3.7 es la siguiente:

```
El valor decimal (base 10) de 15 es 15
El valor octal (base 8) de 15 es 017
El valor hexadecimal (base 16) de 15 es 0xf
```

El despliegue de valores enteros en uno de los tres sistemas numéricos posibles (decimal, octal y hexadecimal) no afecta la manera en que se almacena el número dentro de una computadora. Todos los números se almacenan usando los códigos internos propios de la computadora. Los manipuladores enviados a `cout` le indican al objeto cómo convertir el código interno con el propósito de desplegar la salida.

Además de desplegar enteros en forma octal o hexadecimal, pueden escribirse las constantes en números enteros en un programa en estos formatos. Para designar una constante entera octal, el número debe tener un cero a la izquierda. El número 023, por ejemplo, es un número octal en C++. Los números hexadecimales se denotan usando un 0x a la izquierda. El uso de constantes enteras octales y hexadecimales se ilustra en el programa 3.8.

Programa 3.8

```
#include <iostream>
using namespace std;

int main()
{
  cout << "El valor decimal de 025 es " << 025 << endl;
       << "El valor decimal de 0x37 es "<< 0x37 << endl;

  return 0;
}
```

La salida producida por el programa 3.8 es la siguiente:

```
El valor decimal de 025 es 21
El valor decimal de 0x37 es 55
```

La relación entre la entrada, almacenamiento y despliegue de enteros se ilustra en la figura 3.8.

Por último, los manipuladores especificados en las tablas 3.1 y 3.2 pueden establecerse usando los métodos de clase `ostream` enumerados en la tabla 3.4.

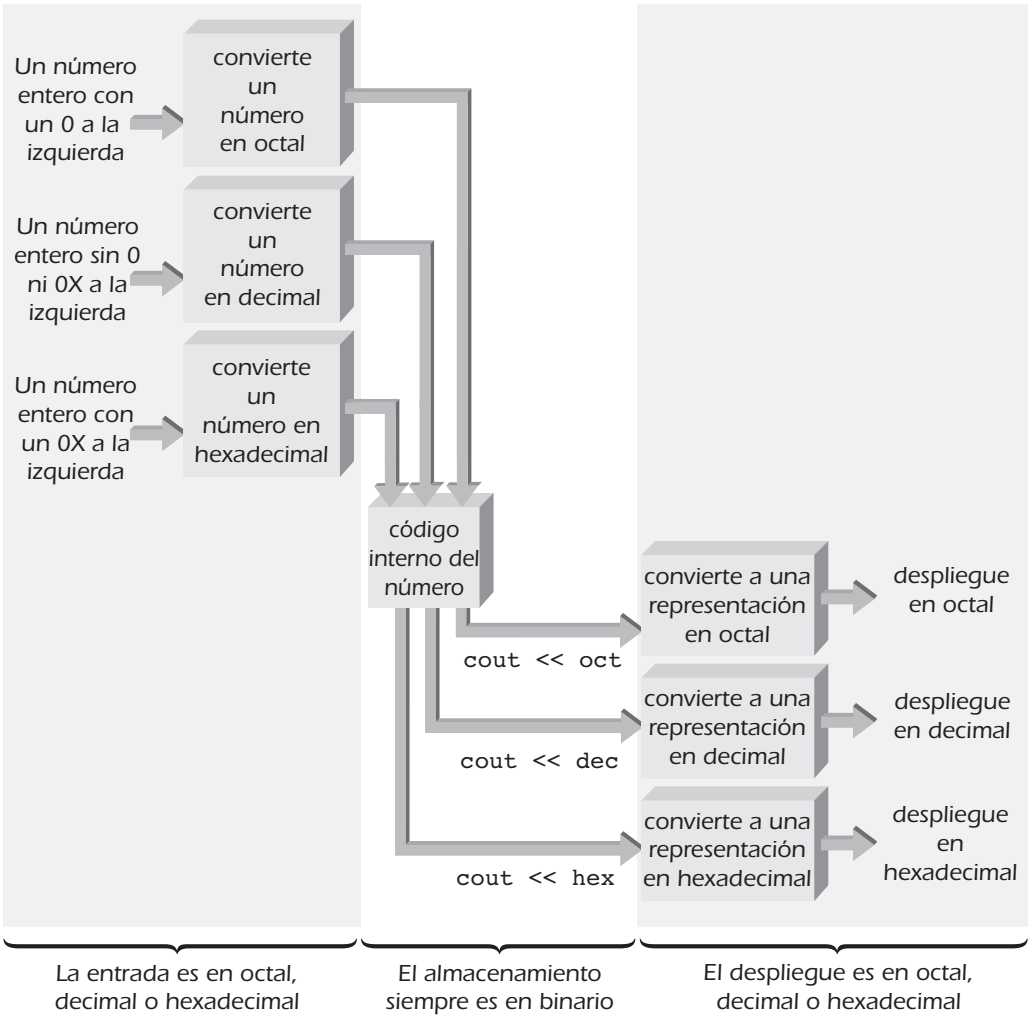


Figura 3.8 *Entrada, almacenamiento y despliegue de números enteros.*

Tabla 3.4 Métodos de clase `ostream`

Método	Comentario	Ejemplo
`precision(n)`	Equivalente a `setprecision()`	`cout.precision(2)`
`fill('x')`	Equivalente a `setfill()`	`cout.fill('*')`
`setf(ios::fixed)`	Equivalente a `setiosflags(ios::fixed)`	`cout.setf(ios::fixed)`
`setf(ios::showpoint)`	Equivalente a `setiosflags(ios::showpoint)`	`cout.setf(ios::showpoint)`
`setf(iof::left)`	Equivalente a `left`	`cout.setf(ios::left)`
`setf(ios::right)`	Equivalente a `right`	`cout.setf(ios::right)`
`setf(ios::flush)`	Equivalente a `endl`	`cout.setf(ios::flush)`

En la columna de ejemplos de la tabla 3.4, el nombre del objeto, `cout`, se separa del método con un punto. Ésta es la forma estándar de llamar a un método de clase y proporcionarle el objeto sobre el que va a operar.

Ejercicios 3.2

1. Determine la salida del siguiente programa:

```
#include <iostream>
using namespace std;

int main()  // un programa que ilustra el truncamiento de
                números enteros
{
  cout << "respuesta1 es el entero " <<  9/4
       << "\nrespuesta2 es el entero " <<  17/3 << endl;

  return 0;
}
```

2. Determine la salida del siguiente programa:

```
#include <iostream>
using namespace std;

int main()  // un programa que ilustra el operador
{
  cout << "El residuo de 9 dividido entre 4 es " <<  9 % 4
       << "\nEl residuo de 17 dividido entre 3 es " << 17 % 3 << endl;

  return 0;
}
```

3. Escriba un programa en C++ que despliegue los resultados de las expresiones 3.0 * 5.0, 7.1 * 8.3 – 2.2 y 3.2 / (6.1 * 5). Calcule el valor de estas expresiones en forma manual para verificar que los valores desplegados son correctos.

4. Escriba un programa en C++ que despliegue los resultados de las expresiones 15 / 4, 15 % 4 y 5 * 3 – (6 * 4). Calcule el valor de estas expresiones en forma manual para verificar que el despliegue producido por su programa es correcto.

5. Determine los errores en cada una de las siguientes instrucciones:

   a. `cout << "\n << " 15)`
   b. `cout << "setw(4)" <<  33;`
   c. `cout << "setprecision(5)" <<  526.768;`
   d. `"Hello World!" >> cout;`
   e. `cout << 47 << setw(6);`
   f. `cout << set(10) << 526.768 << setprecision(2);`

6. Determine y escriba el despliegue producido por las siguientes instrucciones:

   a. `cout << "|" << 5 <<"|";`
   b. `cout << "|" << setw(4) << 5 << "|";`
   c. `cout << "|" << setw(4) << 56829 << "|";`
   d.
```
cout << "|" << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 5.26 << "|";
```
   e.
```
cout << "|" << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 5.267 << "|";
```
   f.
```
cout << "|" << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 53.264 << "|";
```
   g.
```
cout << "|" << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 534.264 << "|";
```
   h.
```
cout << "|" << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 534. << "|";
```

7. Escriba el despliegue producido por las siguientes instrucciones.

   a.
```
cout << "El número es " << setw(6) << setiosflags(ios::fixed)
     << setprecision(2) << 26.27 << endl;
cout << "El número es " << setw(6) << setiosflags(ios::fixed)
     << setprecision(2) << 682.3 << endl;
cout << "El número es " << setw(6) << setiosflags(ios::fixed)
     << setprecision(2) << 1.968 << endl;
```
   b.
```
cout << setw(6) << setiosflags(ios::fixed)
     << setprecision(2) << 26.27 << endl;
cout << setw(6) << setiosflags(ios::fixed)
     << setprecision(2) << 682.3 << endl;
cout << setw(6) << setiosflags(ios::fixed)
     << setprecision(2) << 1.968 << endl;
cout << "------\n";
cout << setw(6) << setiosflags(ios::fixed)
     << setprecision(2)
     << 26.27 + 682.3 + 1.968 << endl;
```

c.
```
cout << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 26.27 << endl;
cout << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 682.3 << endl;
cout << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 1.968 << endl;
cout << "-----\n";
cout << setw(5) << setiosflags(ios::fixed)
     << setprecision(2)
     << 26.27 + 682.3 + 1.968 << endl;
```

d.
```
cout << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 36.164 << endl;
cout << setw(5) << setiosflags(ios::fixed)
     << setprecision(2) << 10.003 << endl;
cout << "-----" << endl;
```

8. La siguiente tabla enumera la correspondencia entre los números decimales 1 a 15 y su representación octal y hexadecimal.

Decimal:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Octal:	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
Hexadecimal:	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

Usando la tabla anterior, determine la salida del siguiente programa.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << "\nEl valor de 14 en octal es " << oct << 14
       << "\nEl valor de 14 en hexadecimal es " << hex << 14
       << "\nEl valor de 0xA en decimal es " << dec << 0xA
       << "\nEl valor de 0xA en octal es " << oct << 0xA
       << endl;

  return 0;
}
```

9. La resistencia combinada de tres resistores conectados en paralelo, como se muestra en la figura 3.9, está dada por la ecuación

$$\text{Resistencia combinada} = \frac{1}{\frac{1}{R_1}+\frac{1}{R_2}+\frac{1}{R_3}}$$

Usando esta fórmula, escriba un programa en C++ para calcular y desplegar la resistencia combinada cuando los tres resistores $R_1 = 1000$, $R_2 = 1000$ y $R_3 = 1000$ están conectados en paralelo. La salida deberá producir el despliegue

```
La resistencia combinada es xxxx.xx ohmios,
```

donde xxxx.xx denota que el valor calculado deberá colocarse en un ancho de campo de 7 columnas, con dos posiciones a la derecha del punto decimal.

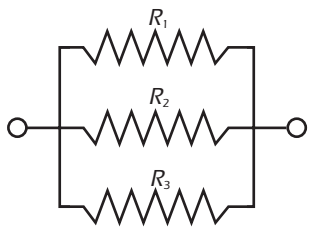


Figura 3.9 *Tres resistores conectados en paralelo.*

10. Escriba un programa en C++ para calcular y desplegar el valor de la pendiente de la línea que conecta los dos puntos cuyas coordenadas son (3, 7) y (8, 12). Use el hecho que la pendiente entre dos puntos que tienen coordenadas (x1, y1) y (x2, y2) es pendiente = (y2 – y1) / (x2 – x1). El despliegue producido por su programa deberá ser: El valor de la pendiente es xxx.xx, donde xxx.xx denota que el valor calculado deberá ser colocado en un ancho de campo suficiente para tres lugares a la izquierda del punto decimal y dos lugares a la derecha de éste.

11. Escriba un programa en C++ para calcular y desplegar las coordenadas del punto medio de la línea que conecta los dos puntos cuyas coordenadas son (3, 7) y (8, 12). Use el hecho que las coordenadas del punto medio entre dos puntos que tienen coordenadas (x1, y1) y (x2, y2) son ((X1 + X2)/2, (Y1 + Y2)/2). El despliegue producido por su programa deberá ser:

```
La coordenada x del punto medio es xxx.xx
La coordenada y del punto medio es xxx.xx
```

donde xxx.xx denota que el valor calculado deberá colocarse en un ancho de campo suficiente para tres lugares a la izquierda del punto decimal y dos lugares a la derecha de éste.

12. Escriba un programa en C++ para calcular y desplegar el momento de flexión máxima, *M*, de una viga, la cual está sostenida en ambos extremos (véase la figura 3.10). La fórmula para el momento de flexión máximo es, $M = XW(L - X) / L$, donde *X* es la distancia del extremo de la viga en que se coloca un peso, *W* y *L* es el largo de la viga. El despliegue producido por su programa deberá ser

```
El momento de flexión máxima es xxxx.xxxx
```

donde xxxx.xxxx denota que el valor calculado deberá colocarse en un ancho de campo suficiente para cuatro lugares a la derecha y a la izquierda del punto decimal.

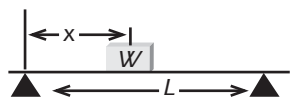


Figura 3.10 *Cálculo del momento de flexión máxima.*

13. Para el circuito eléctrico mostrado en la figura 3.11, las corrientes en los ramales, i_1, i_2 e i_3 pueden determinarse usando las fórmulas

$$i_1 = \frac{E_2R_3 + E_1(R_1 + R_3)}{(R_1 + R_3)(R_2 + R_3) - (R_3)^2}$$

$$i_2 = \frac{E_1R_3 + E_2(R_1 + R_3)}{(R_1 + R_3)(R_2 + R_3) - (R_3)^2}$$

$$i_3 = i_1 - i_2$$

Usando estas fórmulas, escriba un programa en C++ para calcular las corrientes en los ramales cuando R_1 = 10 ohmios, R_2 = 4 ohmios, R_3 = 6 ohmios, E_1 = 12 voltios y E_2 = 9 voltios. El despliegue producido por su programa deberá ser

```
La corriente en el ramal 1 es xx.xxxxx
La corriente en el ramal 2 es xx.xxxxx
La corriente en el ramal 3 es xx.xxxxx
```

donde xx.xxxxx denota que el valor calculado deberá colocarse en un ancho de campo suficiente para dos lugares a la izquierda del punto decimal y cinco lugares a la derecha de éste.

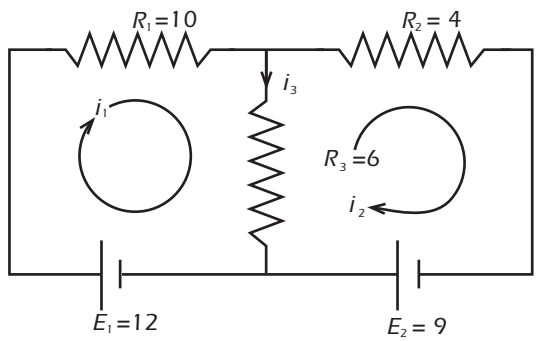


Figura 3.11 *Cálculo de corrientes en circuito en un circuito eléctrico.*

3.3 EMPLEO DE LA BIBLIOTECA DE FUNCIONES MATEMÁTICAS

Como se ha visto, las instrucciones de asignación pueden usarse para ejecutar cálculos aritméticos. Por ejemplo, la instrucción de asignación

```
voltios = resistencia * corriente;
```

multiplica el valor en `corriente` por el valor en `resistencia` y asigna el valor resultante a `voltios`. Aunque la adición, sustracción, multiplicación y división se logran con facilidad usando operadores aritméticos de C++, no existen operadores para elevar un número a una potencia, encontrar la raíz cuadrada de un número o determinar valores trigonométricos.

Para facilitar estos cálculos, C++ proporciona funciones preprogramadas estándares que pueden incluirse en un programa.

Antes de usar una de las funciones matemáticas de C++, necesita saber

- El nombre de la función matemática deseada
- Qué hace la función matemática
- El tipo de datos requerido por la función matemática
- El tipo de datos del resultado devuelto por la función matemática
- Cómo incluir la biblioteca

Para ilustrar el uso de las funciones matemáticas de C++, considere la función matemática llamada `sqrt`, la cual calcula la raíz cuadrada de un número. La raíz cuadrada de un número se calcula usando la expresión

```
sqrt(número)
```

donde el nombre de la función, en este caso `sqrt`, es seguido por paréntesis que contienen el número cuya raíz cuadrada se desea calcular. El propósito de los paréntesis que siguen al nombre de la función es proporcionar un embudo por el que puedan pasar los datos a la función (véase la figura 3.12). Los elementos que pasan a la función por medio de los paréntesis se llaman argumentos de la función y constituyen sus datos de entrada. Por ejemplo, las siguientes expresiones se usan para calcular la raíz cuadrada de los argumentos 4., 17.0, 25., 1043.29 and 6.4516, respectivamente:

```
sqrt(4.)
sqrt(17.0)
sqrt(25.)
sqrt(1043.29)
sqrt(6.4516)
```

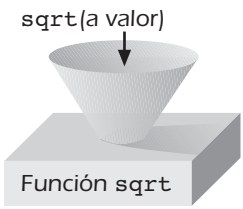


Figura 3.12 *Transmisión de datos a la función* `sqrt()`.

Hay que observar que el argumento para la función `sqrt()` debe ser un valor real. Éste es un ejemplo de las capacidades de sobrecarga de la función de C++. La sobrecarga de la función permite que el mismo nombre de la función sea definido para argumentos con diferentes tipos de datos. En este caso en realidad hay tres funciones de raíz cuadrada nombradas `sqrt()`, una definida para argumentos en número de punto flotante, de doble precisión y de

doble precisión largos. La función `sqrt` correcta es invocada dependiendo del tipo de valor que se le da. La función `sqrt()` determina la raíz cuadrada de su argumento y devuelve el resultado como un número doble. Los valores devueltos por las expresiones anteriores son

Expresión	Valor devuelto
`sqrt(4.)`	2
`sqrt(17.0)`	4.12311
`sqrt(25.)`	5
`sqrt(1043.29)`	32.3
`sqrt(6.4516)`	2.54

Además de la función `sqrt`, la tabla 3.5 enumera las funciones matemáticas de C++ más usadas. Tener acceso a estas funciones en un programa requiere que se incluya con la función el archivo de encabezado matemático llamado `cmath`, el cual contiene declaraciones apropiadas para la función matemática. Esto se hace colocando la siguiente instrucción preprocesadora al principio de cualquier programa que use una función matemática:

```
#include <cmath>
```
← sin punto y coma

Aunque algunas de las funciones matemáticas enumeradas requieren más de un argumento, todas las funciones, por definición, pueden devolver en forma directa como máximo un valor. Además, todas las funciones enumeradas están sobrecargadas: esto significa que puede usarse el mismo nombre de función con argumentos con números enteros y reales. La tabla 3.6 ilustra el valor devuelto por funciones selectas usando argumentos de ejemplo.

Tabla 3.5 Funciones comunes de C++

Nombre de la función	Descripción	Valor devuelto
`abs(a)`	valor absoluto	mismo tipo de datos que el argumento
`pow(a1,a2)`	a1 elevado a la potencia a2	tipo de datos del argumento a1
`sqrt(a)`	raíz cuadrada de un número real	precisión doble
`sin(a)`	seno de a (a en radianes)	doble
`cos(a)`	coseno de a (d en radianes)	doble
`tan(a)`	tangente de a (d en radianes)	doble
`log(a)`	logaritmo natural de a	doble
`log10(a)`	logaritmo común (base 10) de a	doble
`exp(a)`	e elevado a la potencia a	doble

Table 3.6 Ejemplos de funciones selectas

Ejemplo	Valor devuelto
`abs(-7.362)`	7.362
`abs(-3)`	3
`pow(2.0,5.0)`	32
`pow(10,3)`	1000
`log(18.697)`	2.92836
`log10(18.697)`	1.27177
`exp(-3.2)`	0.040762

En cada caso que se usa una función matemática, ésta se activa al dar el nombre de la función y transmitirle datos dentro del paréntesis que sigue al nombre de la función (véase la figura 3.13).

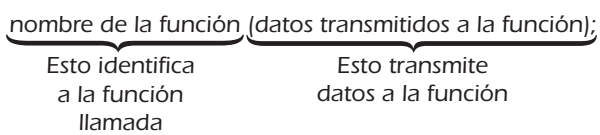


Figura 3.13 *Uso y transmisión de datos a una función.*

Los argumentos que se transmiten a una función no necesitan ser constantes simples. Las expresiones también pueden ser argumentos, siempre que la expresión pueda calcularse para producir un valor del tipo de datos requerido. Por ejemplo, los siguientes argumentos son válidos para las funciones dadas:

```
sqrt(4.0 + 5.3 * 4.0)          abs(2.3 * 4.6)
sqrt(16.0 * 2.0 - 6.7)         sin(theta - phi)
sqrt(x * y - z/3.2)            cos(2.0 * omega)
```

Las expresiones entre paréntesis se evalúan primero para producir un valor específico. Por tanto, tendrían que asignarse valores a las variables theta, phi, x, y, z y omega antes de usarse en las expresiones anteriores. Después que se calcula el valor del argumento, éste se transmite a la función.

Las funciones también pueden incluirse como parte de expresiones más grandes. Por ejemplo,

```
   4 * sqrt(4.5 * 10.0 - 9.0) - 2.0
=  4 * sqrt(36.0) - 2.0
=  4 * 6.0 - 2.0
=  24.0 - 2.0
=  22.0
```

La evaluación paso por paso de una expresión como

```
3.0 * sqrt(5 * 33 - 13.71) / 5
```

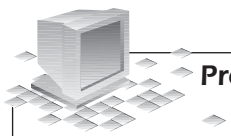
es

Paso	Resultado
1. Realizar la multiplicación en el argumento	`3.0 * sqrt(165 - 13.71) / 5`
2. Completar el cálculo del argumento	`3.0 * sqrt(151.29) / 5`
3. Devolver un valor de la función	`3.0 * 12.3 / 5`
4. Realizar la multiplicación	`36.9 / 5`
5. Realizar la división	`7.38`

El programa 3.9 ilustra el uso de la función `sqrt` para determinar el tiempo que tarda una pelota en golpear el suelo después de haber sido dejada caer desde una torre de 800 pies. La fórmula matemática usada para calcular el tiempo, en segundos, que tarda en caer una distancia determinada, en pies, es

```
tiempo = sqrt(2 * distancia / g)
```

donde *g* es la constante gravitacional igual a 32.2 pies/s^2.



Programa 3.9

```
#include <iostream>  // esta línea puede colocarse en segundo lugar en vez
                        de en primero
#include <cmath>     // esta línea puede colocarse en primer lugar en vez
                        de en segundo
using namespace std;

int main()
{
  int altura;
  double tiempo;

  altura = 800;
  tiempo = sqrt(2 * altura / 32.2);
  cout << "Tardará " << tiempo << " segundos en caer "
       << altura << " pies.\n";

  return 0;
}
```

La salida producida por el programa 3.9 es

```
Tardará 7.04907 segundos en caer 800 pies.
```

Como se usa en el programa 3.9, el valor devuelto por la función `sqrt` es asignado a la variable `tiempo`. Además de asignar el valor devuelto de una función a una variable, el valor devuelto puede incluirse dentro de una expresión más grande, o incluso usarse como un argumento para otra función. Por ejemplo, la expresión

```
sqrt( sin( abs(theta) ) )
```

es válida. En vista que están presentes paréntesis, el cálculo procede de los pares interiores hacia los pares de paréntesis exteriores. Por tanto, el valor absoluto de `theta` se calcula primero y se usa como un argumento para la función sin. El valor devuelto por la función `sin` se usa luego como un argumento para la función `sqrt()`.

Hay que observar que los argumentos de todas las funciones trigonométricas (`sin`, `cos`, etc.) deben expresarse en radianes. Por tanto, para obtener el seno de un ángulo que está dado en grados, primero debe convertirse el ángulo a una medida en radianes. Esto se logra con facilidad al multiplicar el ángulo por el término (3.1416/180.). Por ejemplo, para obtener el seno de 30 grados, puede usarse la expresión `sin (30 * 3.1416/180.)`

Moldes

Ya se ha visto la conversión del tipo de datos de un operando dentro de expresiones aritméticas en modo mixto (sección 2.4) y mediante operadores de asignación (sección 3.1). Además de estas conversiones del tipo de datos implícitas, C++ también proporciona conversiones de tipo explícitas especificadas por el usuario. El operador usado para forzar la conversión de un valor a otro tipo es el **operador de molde** (cast). C++ proporciona operadores de molde en tiempo de compilación y en tiempo de ejecución.

El molde en tiempo de compilación es un operador unitario que tiene la sintaxis `tipoDeDatos(expresión)`, donde `tipoDeDatos` es el tipo de datos deseado al que se convierte la expresión entre paréntesis. Por ejemplo, la siguiente expresión

```
int (a * b)
```

asegura que el valor de la expresión `a * b` es convertido en un valor de número entero.[7]

Con la introducción del estándar más reciente de C++, se incluyeron moldes en tiempo de ejecución. En este tipo de molde, la conversión de tipo solicitada es verificada en tiempo de ejecución y se aplica si la conversión produce un valor válido. Aunque se dispone de cuatro tipos de moldes en tiempo de ejecución, el molde más usado y que corresponde al molde en tiempo de compilación tiene la sintaxis siguiente:

```
staticCast<tipo-de-datos> (expresión)
```

Por ejemplo, el molde en tiempo de ejecución `staticCast<int>(a * b)` es equivalente al molde en tiempo de compilación `int (a* b)`.

[7]La sintaxis de tipo molde en C, en este caso `(int)(a * b)`, también funciona en C++.

Ejercicios 3.3

1. Escriba las llamadas de función para determinar:
   a. La raíz cuadrada de 6.37.
   b. La raíz cuadrada de x – y.
   c. El seno de 30 grados.
   d. El seno de 60 grados.
   e. El valor absoluto de $a^2 - b^2$.
   f. El valor de e elevado a la tercera potencia.

2. Para $a = 10.6$, $b = 13.9$, $c = -3.42$, determine los siguientes valores:
   a. `int (a)`
   b. `int (b)`
   c. `int (c)`
   d. `int (a + b)`
   e. `int (a) + b + c`
   f. `int (a + b) + c`
   g. `int (a + b + c)`
   h. `float (int (a)) + b`
   i. `float (int (a + b))`
   j. `abs(a) + abs(b)`
   k. `sqrt(abs(a - b))`

3. Escriba instrucciones de C++ para lo siguiente:
   a. b = seno x – cos x
   b. $b = \text{seno}^2 x - \cos^2 x$
   c. *área* = (c * b * seno a)/2
   d. $c = \sqrt{a^2 + b^2}$
   e. $p = \sqrt{|m - n|}$
   f. $suma = \dfrac{a\left(r^n - 1\right)}{r - 1}$

4. Escriba, compile y ejecute un programa en C++ que calcule y devuelva la raíz cuarta del número 81.0, la cual es 3. Cuando haya verificado que su programa funciona en forma correcta, úselo para determinar la raíz cuarta de 1,728.896400. Su programa deberá usar la función `sqrt()`.

5. Escriba, compile y ejecute un programa en C++ que calcule la distancia entre dos puntos cuyas coordenadas son (7, 12) y (3, 9). Use el hecho que la distancia entre dos puntos que tienen coordenadas (x_1, y_1) y (x_2, y_2) es *distancia* = $sqrt([x_1 - x_2]^2 + [y_1 - y_2]^2)$. Cuando haya verificado que su programa funciona en forma correcta, calculando la distancia entre los dos puntos en forma manual, use su programa para determinar la distancia entre los puntos (–12, –15) y (22, 5).

6. Si se coloca una escalera de 20 pies en un ángulo de 85 grados sobre un lado de un edificio, como se ilustra en la figura 3.14, la altura a la que la escalera toca el edificio puede calcularse como *altura* = 20 * seno 85°. Calcule esta altura en forma manual y luego escriba, compile y ejecute un programa en C++ que determine y despliegue el valor de la altura. Cuando haya verificado que su programa funciona en forma correcta, úselo para determinar la altura de una escalera de 25 pies colocada en un ángulo de 85 grados.

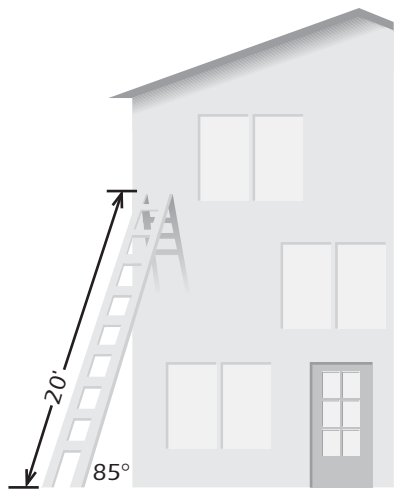


Figura 3.14 Calcular la altura de una escalera contra un edificio.

7. La altura máxima alcanzada por una pelota lanzada con una velocidad inicial v, en metros/segundo, en un ángulo de θ está dada por la fórmula *altura* = (.5 * v^2 * seno^2 θ) / 9.8. Usando esta fórmula, escriba, compile y ejecute un programa en C++ que determine y despliegue la altura máxima alcanzada cuando la pelota es lanzada a 5 millas/hora en un ángulo de 60 grados. (*Sugerencia:* Asegúrese de convertir la velocidad inicial en las unidades correctas. Hay 1609 metros en una milla.) Calcule la altura máxima en forma manual y verifique el resultado producido por su programa. Después de haber verificado que su programa funciona en forma correcta, úselo para determinar la altura alcanzada por una pelota lanzada a 7 millas/hora en un ángulo de 45 grados.

8. Para valores pequeños de x, el valor de seno(x) puede aproximarse con la serie de potencias:

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \cdots$$

Como con la función sin, el valor de x debe expresarse en radianes. Usando esta serie de potencias, escriba, compile y ejecute un programa en C++ que aproxime el seno de 180/3.1416 grados, lo cual es igual a un radián. Además, haga que su programa use la función sin para calcular el seno y desplegar tanto los valores calculados como la diferencia absoluta de los dos resultados. Verifique la aproximación producida por su programa en forma manual. Después que haya verificado que su programa funciona en forma correcta, úselo para aproximar el valor del seno de 62.2 grados.

9. Las coordenadas polares de un punto consisten en la distancia, r, de un origen especificado y un ángulo, θ, con respecto al eje x. Las coordenadas (x y y) del punto se relacionan con sus coordenadas polares por las fórmulas

$$x = r \cos \theta$$
$$y = r \text{ seno } \theta$$

Usando estas fórmulas, escriba un programa en C++ que calcule las coordenadas (x, y) del punto cuyas coordenadas polares son $r = 10$ y $\theta = 30$ grados. Verifique los resultados producidos por su programa calculando los resultados en forma manual. Después de haber verificado que su programa funciona en forma correcta, úselo para convertir las coordenadas polares $r = 12.5$ y $\theta = 67.8°$ en coordenadas rectangulares.

10. Un modelo del crecimiento de la población mundial, en miles de millones de personas, desde 2000 está dado por la ecuación:

$$Población = 6.0\ e^{0.02[año\ -\ 2000]}$$

Usando esta fórmula, escriba, compile y ejecute un programa en C++ para estimar la población mundial en el año 2005. Verifique el resultado desplegado por su programa calculando la respuesta en forma manual. Después que haya verificado que su programa funciona en forma correcta, úselo para estimar la población mundial en el año 2012.

11. Un modelo para estimar el número de gramos de un cierto isótopo radiactivo que restan después de N años está dado por la fórmula

$$Material\ remanente = (material\ original)\ e^{-0.00012N}$$

Usando esta fórmula, escriba, compile y ejecute un programa en C++ para determinar la cantidad de material radiactivo remanente después de 1000 años, suponiendo una cantidad inicial de 100 gramos. Verifique el despliegue producido por su programa usando un cálculo manual. Después que haya verificado que su programa funciona en forma correcta, úselo para determinar la cantidad de material radiactivo remanente después de 275 años, suponiendo una cantidad inicial de 250 gramos.

12. El número de años que se requiere para que se descomponga un cierto isótopo de uranio a la mitad de una cantidad original está dado por la

$$Vida\ media = \ln(2)/k$$

donde k es igual a 0.00012. Usando esta fórmula, escriba, compile y ejecute un programa en C++ que calcule y despliegue la vida media de este isótopo de uranio. Verifique el resultado producido por su programa usando un cálculo manual. Después que haya verificado que su programa funciona en forma correcta, úselo para determinar la vida media de un isótopo de uranio que tenga $k = 0.00026$.

13. La amplificación de circuitos electrónicos se mide en unidades de decibeles, las cuales se calculan como

$$10\ LOG\ (P_o/P_i)$$

donde P_o es la potencia de la señal de salida y P_i es la potencia de la señal de entrada. Usando esta fórmula, escriba, compile y ejecute un programa en C++ que calcule y despliegue la amplificación en decibeles en la que la potencia de salida es 50 veces

la potencia de entrada. Verifique el resultado desplegado por su programa usando un cálculo manual. Después de haber verificado que su programa funciona en forma correcta, úselo para determinar la amplificación de un circuito cuya potencia de salida es 4.639 vatios y la potencia de entrada es 1 vatio.

14. La intensidad de un sonido se mide en unidades de decibeles, las cuales se calculan como

```
10 LOG (SL/RL)
```

donde *SL* es la intensidad del sonido que se está midiendo y *RL* es un sonido de referencia del nivel de intensidad. Usando esta fórmula, escriba un programa en C++ que calcule y despliegue el ruido en decibeles de una calle transitada que tiene una intensidad de sonido de 10 000 000 *RL*. Verifique el resultado producido por su programa usando un cálculo manual. Después que haya verificado que su programa funciona en forma correcta, úselo para determinar el nivel de sonido, en decibeles, de los siguientes sonidos:

a. Un susurro con una intensidad de sonido de 200 *RL*
b. Una banda de rock tocando con una intensidad de sonido de 1 000 000 000 000 *RL*
c. Un avión despegando con una intensidad de sonido de 100 000 000 000 000 *RL*

15. Cuando una pelota de hule especial se deja caer desde una altura dada (en metros) su velocidad de impacto (en metros/segundo) cuando golpea el suelo está dada por la fórmula *velocidad* = sqrt*(2 * g * altura)*. La pelota rebota entonces a 2/3 de la altura desde la cual cayó la última vez. Usando esta información, escriba, pruebe y ejecute un programa en C++ que calcule y despliegue la velocidad de impacto de los primeros tres rebotes y la altura alcanzada en cada rebote. Pruebe su programa usando una altura inicial de 2.0 metros. Ejecute el programa dos veces y compare los resultados de soltar la pelota en la Tierra (g = 9.81 m/s^2) y en la Luna (g = 1.67 m/s^2).

16. a. Una balanza tiene pesas de los siguientes tamaños: 100 lb., 50 lb., 10 lb., 5 lb. y 1 lb. El número de pesas de 100 lb. y 50 lb. requeridas para pesar un objeto cuyo peso es de `PESO` libras puede calcularse usando las siguientes instrucciones de C++:

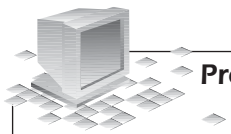
```
// Determine la cantidad de pesas de 100 lb.
      w100 = int(WEIGHT/100)
// Determine la cantidad de pesas de 50 lb.
      w50 = int((WEIGHT - w100 * 100)/50)
```

Usando estas instrucciones como punto de partida, escriba un programa en C++ que calcule la cantidad de cada tipo de pesas necesarias para pesar un objeto de 789 lb.

b. Sin compilar ni ejecutar su programa, compruebe el efecto, en forma manual, de cada instrucción en el programa y determine qué está almacenado en cada variable conforme se encuentra cada instrucción.
c. Cuando haya verificado que su algoritmo funciona en forma correcta, compile y ejecute su programa. Verifique que los resultados producidos por su programa son correctos. Después que haya verificado que su programa funciona en forma correcta, úselo para determinar las pesas requeridas para pesar un objeto de 626 lb.

3.4 ENTRADA DE DATOS AL PROGRAMA USANDO EL OBJETO cin

Los datos para programas que sólo se van a ejecutar una vez pueden incluirse en forma directa en el programa. Por ejemplo, si se desea multiplicar los números 30.0 y 0.05, podría usarse el programa 3.10.



Programa 3.10

```
#include <iostream>
using namespace std;

int main()
{
  double num1, num2, producto;

  num1 = 30.0;
  num2 = 0.05;
  producto = num1 * num2;
  cout << "30.0 por 0.05 es " << producto << endl;

  return 0;
}
```

La salida desplegada por el programa 3.10 es

```
30.0 por 0.05 es 1.5
```

El programa 3.10 puede acortarse, como se ilustra en el programa 3.11. Ambos programas, sin embargo, tienen el mismo problema básico de que deben rescribirse a fin de multiplicar diferentes números. Ambos programas carecen de la facilidad para introducir números diferentes con los cuales operar.

Programa 3.11

```
#include <iostream>
using namespace std;

int main()
{
  cout << "30.0 por 0.05 es " << 30.0 * 0.05 << endl;

  return 0;
}
```

Con excepción de la práctica proporcionada al programador por escribir, introducir y ejecutar el programa, es evidente que los programas que hacen el mismo cálculo sólo una vez, con el mismo conjunto de números, no son muy útiles. Después de todo, es más simple usar una calculadora para multiplicar dos números que introducir y ejecutar el programa 3.10 o 3.11.

Esta sección presenta el objeto `cin`, el cual se usa para introducir datos en un programa mientras se está ejecutando. Del mismo modo que el objeto `cout` despliega una copia del valor almacenado dentro de una variable, el objeto `cin` permite al usuario introducir un valor en la terminal (véase la figura 3.15). El valor se almacena entonces en forma directa en una variable.

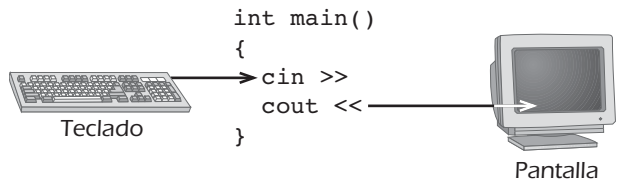


Figura 3.15 `cin` *se usa para introducir datos; cout se usa para desplegar datos.*

Cuando se encuentra una instrucción como `cin >> num1;` la computadora detiene la ejecución del programa y acepta datos del teclado. Cuando se escribe un elemento de datos, el objeto `cin` almacena el elemento en la variable mostrada después del operador de extracción ("obtener de"), `>>`. El programa continúa luego su ejecución con la siguiente instrucción después de la llamada a `cin`. Para ver esto, considere el programa 3.12.

Programa 3.12

```
#include <iostream>
using namespace std;

int main()
{
  double num1, num2, producto;

  cout << "Por favor introduzca un número: ";
  cin  >> num1;
  cout << "Por favor introduzca otro número: ";
  cin  >> num2;
  producto = num1 * num2;
  cout << num1 << " por " << num2 << " es " << producto << endl;

  return 0;
}
```

La primera instrucción `cout` en el programa 3.12 imprime una cadena que le indica a la persona en la terminal qué deberá introducir. Cuando se usa una cadena de salida de esta manera se llama ***indicador de comandos***. En este caso el indicador de comandos le indica al usuario que introduzca un número. La computadora entonces ejecuta la siguiente instrucción, la cual usa un objeto `cin`. El objeto `cin` pone a la computadora en un estado de pausa temporal (o espera) tanto tiempo como le tome al usuario introducir un valor. Luego el usuario le señala al objeto `cin` que se terminó la entrada de datos al oprimir la tecla de retorno después que se ha introducido el valor. El valor introducido se almacena en la variable a la derecha del símbolo de extracción, y la computadora sale de su estado de pausa. Luego procede la ejecución del programa con la siguiente instrucción, la cual en el programa 3.12 es otra instrucción que usa `cout`. Esta instrucción causa que se despliegue el siguiente mensaje. La siguiente instrucción usa entonces `cin` para poner de nuevo a la computadora en un estado de espera temporal mientras el usuario introduce un segundo valor. Este segundo número se almacena en la variable `num2`.

La siguiente ejecución se hizo usando el programa 3.12.

```
Por favor introduzca un número: 30
Por favor introduzca otro número: 0.05
30 por 0.05 es 1.5
```

En el programa 3.12, cada vez que se invoca `cin` se usa para almacenar un valor en una variable. Sin embargo, el objeto `cin` puede usarse para introducir y almacenar tantos valores como símbolos de extracción, `>>`, y variables haya para contener los datos introducidos. Por ejemplo, la instrucción

```
cin >> num1 >> num2;
```

produce dos valores que se leen de la terminal y se asignan a las variables `num1` y `num2.` Si los datos introducidos en la terminal fueran

```
0.052  245.79
```

las variables `num1` y `num2` contendrían los valores 0.052 y 245.79, respectivamente. Hay que observar que cuando se introducen cifras como 0.052 y 245.79 debe haber al menos un espacio entre ellas. El espacio entre las cifras introducidas indica con claridad dónde termina una cifra y comienza la siguiente. Insertar más de un espacio entre cifras no tiene efecto en `cin.`

El mismo espaciado también es aplicable al introducir datos de caracteres; es decir, el operador de extracción, `>>`, se saltará los espacios en blanco y almacenará el siguiente carácter que no sea un espacio en blanco en una variable de carácter. Por ejemplo, en respuesta a las instrucciones

```
char ch1, ch2, ch3;  // declara tres variables de carácter
cin >> ch1 >> ch2 >> ch3;  // acepta tres caracteres
```

La entrada

```
a  b  c
```

causa que la letra `a` sea almacenada en la variable `ch1`, sea almacenada en la variable `ch2,` y la letra `c` sea almacenada en la variable `ch3`. En vista que una variable de carácter sólo puede usarse para almacenar un carácter,

```
abc
```

también puede usarse la entrada.

Puede utilizarse cualquier cantidad de instrucciones que usen el objeto `cin` e introducirse cualquier cantidad de valores usando una sola instrucción `cin`. El programa 3.13 ilustra el uso del objeto `cin` para introducir tres números desde el teclado. El programa calcula luego y despliega el promedio de los números introducidos.

Programa 3.13

```
#include <iostream>
using namespace std;

int main()
{
  int num1, num2, num3;
  double promedio;

  cout << "Introduzca tres números enteros: ";
  cin >> num1 >> num2 >> num3;
  promedio =  (num1 + num2 + num3) / 3.0;
  cout << "El promedio de los números es " << promedio << endl;

  return 0;
}
```

La siguiente muestra de ejecución se hizo usando el programa 3.13:

```
Introduzca tres números enteros: 22 56 73
El promedio de los números es 50.3333
```

Hay que observar que los datos introducidos en el teclado para esta muestra de ejecución consisten en la entrada

```
22 56 73
```

En respuesta a este flujo de entrada, el programa 3.13 almacena el valor 22 en la variable `num1`, el valor 56 en la variable `num2`, y el valor 73 en la variable `num3` (véase la figura 3.16). En vista que el promedio de tres números enteros puede ser un número de punto flotante, la variable promedio, la cual se usa para almacenar el promedio, es declarada como una variable de punto flotante. Hay que observar también que los paréntesis son necesarios en la instrucción de asignación `promedio = (num1 + num2 + num3) / 3.0;`. Sin estos paréntesis, el único valor que se dividiría entre tres sería el entero en num3 (porque la división tiene una precedencia mayor que la adición).

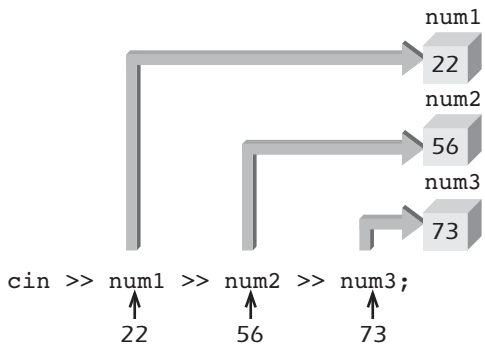


Figura 3.16 *Introducción de datos en las variables* `num1`, `num2`, *and* `num3`.

La operación de extracción `cin` como la operación de inserción `cout` es lo bastante "lista" para hacer unas cuantas conversiones de tipo de datos. Por ejemplo, si se introduce un número entero en lugar de un número de precisión doble, el entero será convertido al tipo de datos correcto.[8] Del mismo modo, si se introduce un número de precisión doble cuando se espera un entero, sólo se usará la parte entera del número. Por ejemplo, suponga que se introducen los siguientes números en respuesta a la instrucción `cin >> num1 >> num2 >> num3;`, donde `num1` y `num3` han sido declarados como variables de precisión doble y `num2` es una variable entera:

```
56  22.879  33.923
```

El *56* será convertido en 56.0 y almacenado en la variable `num1`. La operación de extracción continúa extrayendo datos del flujo de entrada que se le envía, esperando un valor entero. Por lo que respecta a `cin` el punto decimal después de 22 en el número 22.879 indica el final de un entero y el inicio de un número decimal. Por tanto, el número 22 es asignado a `num2`. Al continuar procesando su flujo de entrada, `cin` toma .879 como el siguiente número de punto flotante y lo asigna `num3`. Por lo que respecta a `cin` es entrada extra y es ignorado. Sin embargo, si al principio no se introducen suficientes datos, el objeto `cin` continuará haciendo que la computadora esté en pausa hasta que se hayan introducido suficientes datos.

Una primera mirada a la validación de entradas del usuario

Un programa bien construido debería validar las entradas del usuario y asegurar que no se caiga el programa o produzca una salida sin sentido debido a una entrada inesperada. El término *validar* significa verificar que el valor introducido corresponde al tipo de datos de la variable a la que es asignado el valor dentro de una instrucción `cin` y también que el valor está dentro de un rango aceptable de valores apropiados para la aplicación. Los programas que detectan y responden en forma efectiva a una entrada inesperada del usuario se conocen de manera formal como **programas robustos** y de manera informal como programas "a prueba de balas". Una de sus labores como programador es producir tales programas. Tal como están escritos, los programas 3.12 y 3.13 no son programas robustos. Veamos por qué.

[8]En sentido estricto, lo que viene del teclado no es ningún tipo de datos, como un `int` o `double`, sino es tan sólo una secuencia de caracteres. La operación de extracción maneja la conversión desde secuencia de caracteres hacia un tipo de datos definido.

El primer problema con estos programas se hace evidente cuando un usuario introduce un valor no numérico. Por ejemplo, considere la siguiente muestra de ejecución usando el programa 3.13:

```
Introduzca tres números enteros: 10 20.68 20
El promedio de los números es -2.86331e+008
```

Esta salida ocurre debido a que la conversión del segundo número introducido produce que el valor entero 20 sea asignado a `num2` y el valor –858993460 sea asignado a `num3`.[9] Este último valor corresponde a un carácter inválido, el punto decimal, al que se le asigna un valor entero esperado. El promedio de los números 10, 20 y –858993460 es calculado en forma correcta como –286331143.3, el cual es desplegado en notación científica con seis dígitos significativos como –2.86331e+08. Por lo que respecta al usuario de promedio, esto se reportará como un error del programa. Este mismo problema ocurre siempre que se introduce un valor no entero en cualquiera de las primeras dos entradas. (No ocurre para cualquier valor numérico introducido como la tercera entrada porque la parte entera de la última entrada es aceptada y se ignora la entrada restante.) Como programador, su respuesta inicial puede ser "El programa pide en forma clara que se introduzcan valores enteros". Ésta, sin embargo, es la respuesta de un programador inexperto. Los programadores profesionales entienden que es su responsabilidad asegurar que un programa anticipe y maneje en forma apropiada todas las entradas que introducirá un usuario. Esto se logra pensando qué puede salir mal con su programa mientras lo desarrolla y luego haciendo que otra persona o grupo lo pruebe.

En enfoque básico para manejar la introducción de datos inválidos se conoce como ***validación de entradas del usuario***, lo cual significa validar los datos introducidos durante la introducción o inmediatamente después que los datos han sido introducidos y luego proporcionarle al usuario una forma de reintroducir cualesquiera datos inválidos. La validación de entradas del usuario es una parte esencial de cualquier programa viable desde el punto de vista comercial; si se hace en forma correcta, protege al programa de intentar procesar datos que pueden causar problemas de cálculo. Se verá cómo proporcionar este tipo de validación después de presentar las instrucciones de selección y repetición de C++ en los capítulos 4 y 5, respectivamente.

Ejercicios 3.4

1. Para las siguientes instrucciones de declaración, escriba una instrucción con el objeto `cin` que cause que la computadora entre en pausa mientras el usuario introduce los datos apropiados.

   a.
   ```
   int primernum;
   ```
   b.
   ```
   double calificacion;
   ```
   c.
   ```
   double segundonum;
   ```
   d.
   ```
   char valorclave;
   ```
   e.
   ```
   int mes, anios;
   double promedio;
   ```
   f.
   ```
   char ch;
   int num1,num2;
   double calificacion1, calificacion2;
   ```
   g.
   ```
   double interés, principal, capital;
   double precio, rédito;
   ```

[9] Algunos sistemas aceptarán el .68 como la tercera entrada. En todos los casos el último valor 20 es ignorado.

h.
```
char ch, letra1, letra2;
int num1,num2,num3;
```
i.
```
double temp1,temp2,temp3;
double voltios1, voltios2;
```

2. a. Escriba un programa en C++ que despliegue primero el siguiente indicador:

```
Introduzca la temperatura en grados Celsius:
```

Haga que su programa acepte un valor introducido desde el teclado y convierta la temperatura introducida a grados Fahrenheit, usando la fórmula *Fahrenheit* = (9.0 / 5.0) * *Celsius* + 32.0. Su programa deberá desplegar entonces la temperatura en grados Fahrenheit, usando un mensaje de salida apropiado.

b. Compile y ejecute el programa escrito para el ejercicio 2a. Verifique su programa calculando, en forma manual y luego usando su programa, el equivalente en Fahrenheit de los siguientes datos de prueba:

Conjunto de datos de prueba 1: 0 grados Celsius
Conjunto de datos de prueba 2: 50 grados Celsius
Conjunto de datos de prueba 3: 100 grados Celsius

Cuando esté seguro que su programa funciona en forma correcta, úselo para completar la siguiente tabla:

Celsius	Fahrenheit
45	
50	
55	
60	
65	
70	

3. Escriba, compile y ejecute un programa en C++ que despliegue el siguiente indicador:

```
Introduzca el radio de un círculo:
```

Después de aceptar un valor para el radio, su programa deberá calcular y desplegar el área del círculo. (*Sugerencia: área = 3.1416 * radio*^{2}) Con propósitos de prueba, verifique su programa usando una entrada de prueba de un radio de 3 pulgadas. Des-

pués de determinar en forma manual que el resultado producido por su programa es correcto, use su programa para completar la siguiente tabla:

Radio (pulg.)	Área (pulg.^{2})
1.0	
1.5	
2.0	
2.5	
3.0	
3.5	

4. *a.* Escriba, compile y ejecute un programa en C++ que despliegue los siguientes indicadores:

```
Introduzca las millas recorridas:
Introduzca los galones de gasolina consumidos:
```

Después que se despliegue cada indicador, su programa deberá usar una instrucción `cin` para aceptar datos desde el teclado para el indicador desplegado. Después que se haya introducido el número de galones de gasolina consumidos, su programa deberá calcular y desplegar las millas por galón obtenidas. Este valor deberá ser incluido en un mensaje apropiado y calculado usando a ecuación *millas por galón = millas/galones* consumidos. Verifique su programa usando los siguientes datos de prueba:

Conjunto de datos de prueba 1: Millas = 276, Gasolina = 10 galones
Conjunto de datos de prueba 2: Millas = 200, Gasolina = 15.5 galones

Cuando haya completado su verificación, use su programa para completar la siguiente tabla:

Millas recorridas	Galones consumidos	MPG
250	16.00	
275	18.00	
312	19.54	
296	17.39	

b. Para el programa escrito para el ejercicio 4a, determine cuántas ejecuciones de verificación se requieren para asegurar que el programa funciona en forma correcta y dé una razón que apoye su respuesta.

5. a. Escriba, compile y ejecute un programa en C++ que despliegue los siguientes indicadores:

```
Introduzca un número:
Introduzca un segundo número:
Introduzca un tercer número:
Introduzca un cuarto número:
```

Después que se despliega cada indicador, su programa deberá usar una instrucción cin para aceptar un número desde el teclado para el indicador desplegado. Después que se ha introducido el cuarto número, su programa deberá calcular y desplegar el promedio de los números. El promedio deberá incluirse en un mensaje apropiado. Verifique el promedio desplegado por su programa usando los siguientes datos de prueba:

Conjunto de datos de prueba 1: 100, 100, 100, 100
Conjunto de datos de prueba 2: 100, 0, 100, 0

Cuando haya completado su verificación, use su programa para completar la siguiente tabla:

Números	Promedio
92, 98, 79, 85	
86, 84, 75, 86	
63, 85, 74, 82	

b. Repita el ejercicio 5a, asegurándose que usa el mismo nombre de variable, numero, para cada entrada de número. También use la variable `suma` para la suma de los números. (*Sugerencia:* para hacer esto, puede usar la instrucción `suma = suma + numero` después que se ha aceptado cada número. Repase el material sobre acumulación presentado en la sección 2.3.)

6. a. Escriba, compile y ejecute un programa en C++ que calcule y despliegue el valor de polinomio de segundo orden $ax^2 + bx + c$ para valores introducidos por el usuario de los coeficientes a, b, c y la variable x. Haga que su programa despliegue primero un mensaje informando al usuario que realizará el programa, y luego despliegue indicadores apropiados para avisar al usuario que introduzca los datos deseados. (*Sugerencia:* use un indicador como `Introduzca el coeficiente del término x al cuadrado:`)

b. Verifique el resultado producido por su programa para el ejercicio 6a usando los siguientes datos de prueba:

Conjunto de datos de prueba 1: $a = 0$, $b = 0$, $c = 22$, $x = 56$
Conjunto de datos de prueba 2: $a = 0$, $b = 22$, $c = 0$, $x = 2$
Conjunto de datos de prueba 3: $a = 22$, $b = 0$, $c = 0$, $x = 2$
Conjunto de datos de prueba 4: $a = 2$, $b = 4$, $c = 5$, $x = 2$
Conjunto de datos de prueba 5: $a = 5$, $b = -3$, $c = 2$, $x = 1$

Cuando haya completado su verificación, use su programa para completar la siguiente tabla:

a	b	c	x	valor del polinomio
2.0	17.0	–12.0	1.3	
3.2	2.0	15.0	2.5	
3.2	2.0	15.0	–2.5	
–2.0	10.0	0.0	2.0	
–2.0	10.0	0.0	4.0	
–2.0	10.0	0.0	5.0	
–2.0	10.0	0.0	6.0	
5.0	22.0	18.0	8.3	
4.2	–16	–20	–5.2	

7. El número de bacterias, B, en un cierto cultivo que es sometido a refrigeración puede aproximarse por la ecuación $B = 300000\ e^{-0.032t}$, donde e es el número irracional 2.71828 (redondeado a cinco lugares decimales), conocido como número de Euler, y t es el tiempo, en horas, que se ha refrigerado el cultivo. Usando esta ecuación, escriba, compile y ejecute un programa simple en C++ que indique al usuario que introduzca un valor de tiempo, calcule el número de bacterias en el cultivo y despliegue el resultado. Con propósitos de prueba, verifique su programa usando una entrada de prueba de 10 horas. Cuando haya verificado la operación de su programa, úselo para determinar el número de bacterias en el cultivo después de 12, 18, 24, 36, 48 y 72 horas.

8. Escriba, compile y ejecute un programa que calcule y despliegue el valor de la raíz cuadrada de un número real introducido por el usuario. Verifique su programa calculando las raíces cuadradas de los siguientes datos: 25, 16, 0 y 2. Cuando complete su verificación, use su programa para determinar la raíz cuadrada de 32.25, 42, 48, 55, 63 y 79.

9. Escriba, compile y ejecute un programa que calcule y despliegue la raíz cuarta de un número introducido por el usuario. Recuerde del álgebra elemental que la raíz cuarta de un número puede encontrarse elevando el número a la potencia ¼. (*Sugerencia:* no utilice la división de números enteros; ¿puede ver por qué?) Verifique su programa calculando la raíz cuarta de los siguientes datos: 81, 16, 1 y 0. Cuando haya completado su verificación, use su programa para determinar la raíz cuarta de 42, 121, 256, 587, 1240 y 16 256.

10. Para el circuito en serie mostrado en la figura 3.17, la baja del voltaje, V_2, a través del resistor, R_2, y de la potencia, P_2, enviada al resistor está dada por las ecuaciones $V_2 = I\ R_2$ y $P_2 = I\ V_2$, donde $I = E\ /(R_1 + R_2)$. Usando estas ecuaciones, escriba, compile y ejecute un programa en C++ que indique al usuario que introduzca los valores de E,

R_1 y R_2, calcule la baja de voltaje y la potencia enviadas a R_2, y despliegue los resultados. Verifique su programa usando los datos de prueba: E = 10 voltios, R_1 = 100 ohmios y R_2 = 200 ohmios. Cuando haya completado su verificación, use su programa para completar la siguiente tabla:

E (voltios)	R_1 (ohmios)	R_2 (ohmios)	Baja de voltaje (voltios)	Potencia enviada (vatios)
10	100	100		
10	100	200		
10	200	200		
20	100	100		
20	100	200		
20	200	200		

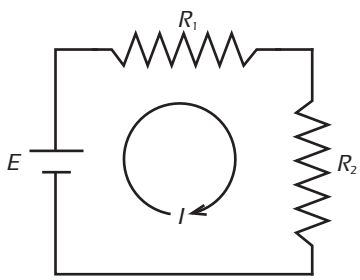


Figura 3.17 *Cálculo de la baja de voltaje.*

11. Escriba, compile y ejecute un programa en C++ que calcule la resistencia combinada de tres resistores paralelos. Los valores de cada resistor deberían ser aceptados usando una instrucción `cin` (use la fórmula para la resistencia combinada dada en el ejercicio 9 de la sección 3.2). Verifique la operación de su programa usando los siguientes datos de prueba:

Conjunto de datos de prueba 1: R_1 = 1000, R_2 = 1000 y R_3 = 1000.
Conjunto de datos de prueba 2: R_1 = 1000, R_2 = 1500 y R_3 = 500.

Cuando haya completado su verificación, use su programa para completar la siguiente tabla:

R1 (voltios)	R2 (ohmios)	R3 (ohmios)	Resistencia combinada (ohmios)
3000	3000	3000	
6000	6000	6000	
2000	3000	1000	
2000	4000	5000	
4000	2000	1000	
10000	100	100	

12. Usando instrucciones `input`, escriba, compile y ejecute un programa en C++ que acepte las coordenadas (*x* y *y*) de dos puntos. Haga que su programa determine y despliegue los puntos medios de los dos puntos (use la fórmula dada en el ejercicio 11 de la sección 3.2). Verifique su programa usando los siguientes datos de prueba:

Conjunto de datos de prueba 1: Punto 1 = (0, 0) y Punto 2 = (16, 0)
Conjunto de datos de prueba 2: Punto 1 = (0, 0) y Punto 2 = (0, 16)
Conjunto de datos de prueba 3: Punto 1 = (0, 0) y Punto 2 = (–16, 0)
Conjunto de datos de prueba 4: Punto 1 = (0, 0) y Punto 2 = (0, –16)
Conjunto de datos de prueba 5: Punto 1 = (–5, –5) y Punto 2 = (5, 5)

Cuando haya completado su verificación, use su programa para completar la siguiente tabla.

Punto 1	Punto 2	Punto medio
(4, 6)	(16, 18)	
(22, 3)	(8, 12)	
(–10, 8)	(14, 4)	
(–12, 2)	(14, 3.1)	
(3.1, –6)	(20, 16)	
(3.1, –6)	(–16, –18)	

13. Escriba, compile y ejecute un programa en C++ que calcule y despliegue el valor del flujo de corriente a través de un circuito RC. El circuito consiste en una batería que está conectada en serie a un interruptor, un resistor y un capacitor. Cuando el interruptor se cierra, la corriente, *i*, que fluye a través del circuito está dada por la ecuación:

$$i = (E/R)\, e^{-t/\tau}$$

donde E es el voltaje de la batería (en voltios), R es la resistencia (en ohmios), τ es la constante de tiempo y t es el tiempo (en segundos) desde que el interruptor fue cerrado.

El programa deberá indicar al usuario que introduzca valores apropiados y use instrucciones de entrada para aceptar los datos. Al construir los indicadores, use instrucciones como `Introduzca el voltaje de la batería`. Verifique la operación de su programa calculando, en forma manual, la corriente para los siguientes datos de prueba:

Conjunto de datos de prueba 1: Voltaje = 20 voltios, R = 10 ohmios, τ = 0.044, t = 0.023 segundos.

Conjunto de datos de prueba 2: Voltaje = 35 voltios = 35, R = 10 ohmios, τ= 0.16, t = 0.067 segundos.

Cuando haya completado su verificación, use su programa para determinar el valor de la corriente para los siguientes casos:

a. Voltaje = 35 voltios, R = 10 ohmios, τ = 0.16, t = 0.11 segundos.
b. Voltaje = 35 voltios, R = 10 ohmios, τ = 0.16, t = 0.44 segundos.
c. Voltaje = 35 voltios, R = 10 ohmios, τ = 0.16, t = 0.83 segundos.
d. Voltaje = 15 voltios, R = 10 ohmios, τ = 0.55, t = 0.11 segundos.
e. Voltaje = 15 voltios, R = 10 ohmios, τ = 0.55, t = 0.44 segundos.
f. Voltaje = 15 voltios, R = 10 ohmios, τ = 0.55, t = 0.067 segundos.
g. Voltaje = 6 voltios, R = 1000 ohmios, τ = 2.6, t = 12.4 segundos.

14. El programa 3.12 indica al usuario que introduzca dos números, donde el primer valor introducido es almacenado en `num1` y el segundo valor es almacenado en `num2`. Usando este programa como punto de partida, escriba un programa que intercambie los valores almacenados en las dos variables.

15. Escriba un programa en C++ que indique al usuario que introduzca un número. Haga que su programa acepte el número como un entero y lo despliegue de inmediato usando una llamada al objeto `cout`. Ejecute su programa tres veces. La primera vez que ejecute el programa introduzca un número entero válido, la segunda vez introduzca un número de precisión doble y la tercera vez introduzca un carácter. Usando el despliegue de salida, vea qué número aceptó en realidad su programa de los datos que introdujo.

16. Repita el ejercicio 15 pero haga que su programa declare la variable usada para almacenar el número como una variable de precisión doble. Ejecute el programa tres veces. La primera vez introduzca un entero, la segunda vez introduzca un número de precisión doble y la tercera vez introduzca un carácter. Usando el despliegue de salida, siga la pista de cuál número aceptó en realidad su programa de los datos que introdujo. ¿Qué sucedió, y por qué?

17. a. ¿Por qué cree que los programas de aplicaciones exitosos contienen verificaciones de validez extensas de los datos de entrada? (*Sugerencia:* revise los ejercicios 16 y 17.)
b. ¿Cuál piensa que es la diferencia entre una verificación del tipo de datos y una verificación de lo razonable que son esos datos?
c. Suponga que un programa requiere que el usuario introduzca un día, mes y año. ¿Cuáles son algunas verificaciones que podría hacer en los datos introducidos?

3.5 CONSTANTES SIMBÓLICAS

Ciertas constantes usadas dentro de un programa tienen un significado más general que es reconocido fuera del contexto del programa. Los ejemplos de estos tipos de constantes incluyen el número 3.1416, el cual es π con una precisión de cuatro lugares decimales; 32.2 pies/sec^2, lo cual es la constante gravitacional; y el número 2.71828, el cual es el número de Euler con una precisión de cinco lugares decimales.

El significado de otras constantes que aparecen en un programa se define estrictamente dentro del contexto de la aplicación que se está programando. Por ejemplo, al determinar el peso de objetos de varios tamaños, la densidad del material que se está usando adquiere un significado especial. Por sí mismos los números de densidad son bastante ordinarios, pero en el contexto de una aplicación particular tienen un significado especial. Números como éstos son conocidos a veces por los programadores como ***números mágicos***. Cuando el mismo número mágico aparece de manera repetida dentro del mismo programa se vuelve una fuente potencial de error, por lo que se tendrá que cambiar la constante. Sin embargo, múltiples cambios están sujetos a error, con un solo valor que se pase por alto y no sea cambiado, el resultado obtenido cuando se ejecuta el programa será incorrecto y la fuente de error difícil de localizar.

Para evitar el problema de tener un número mágico diseminado en un programa en muchos lugares y permitir la identificación clara de constantes universales, como π, C++ permite al programador darle a estas constantes su propio nombre simbólico. Entonces, en lugar de usar el número en todo el programa, se usa en cambio un nombre simbólico. Si alguna vez se tiene que cambiar el número, el cambio sólo necesita hacerse una vez en el punto donde el nombre simbólico es equiparado con el valor numérico real. Equiparar números con nombres simbólicos se logra usando el calificador de declaración `const`. El calificador `const` especifica que el identificador declarado sólo puede leerse después que es inicializado; no puede cambiarse. Tres ejemplos que usan este calificador son

```
const double PI = 3.1416;
const double DENSIDAD = 0.238;
const int MAXNUM = 100;
```

La primera instrucción de declaración crea una constante de doble precisión llamada PI y la inicializa con el valor 3.1416, mientras la segunda instrucción de declaración crea la constante de precisión doble llamada DENSIDAD y la inicializa con 0.238. Por último, la tercera declaración crea una constante entera llamada MAXNUM y la inicializa con el valor 100.

Una vez que se crea e inicializa un identificador `const`, el valor almacenado en él no puede cambiarse. Por tanto, para propósitos prácticos, el nombre de la constante y su valor se vinculan por la duración del programa que los declara.

Aunque hemos escrito los identificadores `const` en letras mayúsculas, podrían haberse usado letras minúsculas. Sin embargo, es común en C++, usar letras mayúsculas para los identificadores `const` a fin de identificarlos con facilidad. Luego, siempre que un programador vea letras mayúsculas en un programa, sabrá que el valor de la constante no puede cambiarse.

Una vez declarado, puede usarse un identificador `const` en cualquier instrucción C++ en lugar del número que representa. Por ejemplo, las instrucciones de asignación

```
circum = 2 * PI * radio;
peso = DENSIDAD * volumen;
```

son válidas. Por supuesto que estas instrucciones deben aparecer después de las declaraciones para todas sus variables. En vista que una declaración `const` equipara de manera efectiva un valor constante con un identificador, y el identificador puede ser usado como un reemplazo directo para su constante inicializada, estos identificadores se conocen por lo general como ***constantes simbólicas*** o ***constantes nombradas***. Se usarán estos términos en forma intercambiable.

Colocación de instrucciones

En esta etapa se han introducido una variedad de tipos de instrucciones. La regla general en C++ para la colocación de las instrucciones es tan sólo que una variable o constante simbólica debe declararse antes que pueda ser usada. Aunque esta regla permite que tanto las directivas del preprocesador como las instrucciones de declaración sean colocadas a lo largo de un programa, hacerlo así resultará en una estructura de programa muy pobre. Como una buena forma de programación, deberá usarse el siguiente orden en las instrucciones:

```
directivas del preprocesador

int main()
{
  constantes simbólicas
  declaraciones de la función principal

  otras instrucciones ejecutables

  return valor
}
```

Conforme se introduzcan nuevos tipos de instrucción se expandirá esta estructura de colocación para acomodarlos. Hay que observar que las instrucciones de comentario pueden entremezclarse con libertad en cualquier parte dentro de esta estructura básica.

El programa 3.14 ilustra esta estructura básica y usa una constante simbólica para calcular el peso de un cilindro de acero. La densidad del acero es 0.284 lb/pulg^3.

Programa 3.14

```
// Este programa determina el peso de un cilindro de acero
//   al multiplicar el volumen del cilindro por su densidad
//  El volumen del cilindro está dado por la fórmula PI * pow(radio,2) * altura.
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  const double PI = 3.1416;
  const double DENSIDAD = 0.284;
  double radio, altura, peso;

  cout << "Introduzca el radio del cilindro (en pulgadas): ";
  cin  >> radio;
  cout << "Introduzca la altura del cilindro (en pulgadas): ";
  cin  >> altura;
  weight = DENSIDAD * PI * pow(radio,2) * altura;
  cout << setiosflags(ios:: fixed)
         << setiosflags(ios::showpoint)
         << setprecision(4)
         << "El cilindro pesa " << peso << " libras" << endl;

  return 0;
}
```

Se puede observar en el programa 3.14 que se han definido dos constantes simbólicas: PI y DENSIDAD. La siguiente ejecución se hizo para determinar el peso de un cilindro con un radio de 3 pulgadas y una altura de 12 pulgadas.

```
Introduzca el radio del cilindro (en pulgadas): 3
Introduzca la altura del cilindro (en pulgadas): 12
El cilindro pesa 96.3592 libras
```

La ventaja de usar la constante nombrada PI en el programa 3.14 es que identifica con claridad el valor de 3.1416 en términos reconocibles por la mayoría de las personas. La ventaja de usar la constante nombrada DENSIDAD es que permite que el programador cambie el valor de la densidad por otro material sin tener que buscar por todo el programa para ver dónde se usa la densidad. Por supuesto, si van a ser considerados muchos materiales diferentes, la densidad deberá cambiarse de una constante simbólica a una variable. Surge una interrogante natural, entonces, sobre la diferencia entre constantes simbólicas y variables.

El valor de una variable puede alterarse en cualquier parte dentro de un programa. Por su naturaleza, una constante nombrada es un valor constante que no debe alterarse después que se ha definido. Nombrar una constante en lugar de asignar el valor a una variable asegura que el valor en la constante no pueda ser alterada en lo subsiguiente. Siempre que aparece una constante nombrada en una instrucción tiene el mismo efecto que la constante que representa. Por tanto, DENSIDAD en el programa 3.14 tan sólo es otra forma de representar el número 0.284. En vista que DENSIDAD y el número 0.284 son equivalentes, el valor de DENSIDAD no puede cambiarse después dentro del programa. Una vez que se ha definido DENSIDAD como una constante, una instrucción de asignación como

```
DENSIDAD = 0.156;
```

carece de significado y producirá un mensaje de error, porque DENSIDAD no es una variable. En vista que DENSIDAD es sólo un sustituto para el valor 0.284, esta instrucción es equivalente a escribir la expresión inválida 0.284 = 0.156. Además de usar una instrucción `const` para nombrar constantes, como en el programa 3.14, esta instrucción también puede utilizarse para equiparar el valor de una expresión constante con un nombre simbólico. Una expresión constante es una expresión que consta sólo de operadores y constantes. Por ejemplo, la instrucción

```
const double GRAD_A_RAD = 3.1416/180.0;
```

equipara el valor de la expresión constante 3.1416/180.0 con el nombre simbólico `GRAD_A_RAD`. El nombre simbólico, como siempre, puede ser usado en cualquier instrucción después de su definición. Por ejemplo, en vista que la expresión 3.1416/180.0 se requiere para convertir grados a radianes, el nombre simbólico seleccionado para este factor de conversión puede usarse en forma conveniente siempre que se requiera una conversión de este tipo. Por tanto, en la instrucción de asignación

```
altura = distancia * sin(angulo * GRAD_A_RAD);
```

la constante simbólica `GRAD_A_RAD` se usa para convertir el valor de la medida de un ángulo en radianes.

Una constante nombrada definida con anterioridad también puede usarse en una instrucción `const` subsiguiente. Por ejemplo, la siguiente secuencia de instrucciones es válida:

```
const double PI = 3.1416;
const double  GRAD_A_RAD = PI / 180.0;
```

En vista que la constante 3.1416 había sido equiparada con el nombre simbólico PI, puede usarse de manera legítima en cualquier definición subsiguiente, aun dentro de otra instrucción `const`. El programa 3.15 usa la constante nombrada `GRAD_A_RAD` para convertir un ángulo introducido por un usuario, en grados, en su medida equivalente en radianes para que la use la función `sin`.

Programa 3.15

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  const double PI = 3.1416;
  const double GRAD_A_RAD = PI/180.0;
  double angulo;

  cout << "Introduzca el ángulo (en grados): ";
  cin >> angulo;
  cout << setiosflags(ios:: fixed)
       << setiosflags(ios::showpoint)
       << setprecision(4)
       << "El seno del ángulo es " << sin(angulo * GRAD_A_RAD) << endl;

  return 0;
}
```

La siguiente muestra de ejecución se hizo usando el programa 3.15.

```
Introduzca el ángulo (en grados): 30
El seno del ángulo es 0.5000
```

Aunque se ha usado el calificador `const` para construir constantes simbólicas, este tipo de datos se encontrará una vez más en el capítulo 6, donde se mostrará que son útiles como argumentos de función al asegurar que el argumento no es modificado dentro de la función.

Ejercicios 3.5

1. Modifique el programa 3.9 para usar la constante nombrada GRAV en lugar del valor 32.2 usado en el programa. Compile y ejecute su programa para verificar que produce el mismo resultado mostrado en el texto.
2. Vuelva a escribir el siguiente programa para usar la constante nombrada FACTOR en lugar de la expresión (5.0/9.0) contenida dentro del programa.

```
#include <iostream>
using namespace std;

int main()
```

```
{
  double fahren, celsius;
  cout << "Introduzca una temperatura en grados Fahrenheit: ";
  cin  >> fahren;
  celsius = (5.0/9.0) * (fahren - 32.0);
  cout << "La temperatura Celsius equivalente es "
       << celsius << endl;

  return 0;
}
```

3. Vuelva a escribir el siguiente programa para usar la constante simbólica PRIMA en lugar del valor 0.04 contenido dentro del programa.

```
#include <iostream>
using namespace std;

int main()
{
  float prima, cantidad, interés;
  prime = 0.04;      // tasa de interés de la prima
  cout << <Introduzca la cantidad: ";
  cin  >> cantidad;
  interés = prima * cantidad;
  cout << "El interés ganado es"
       << interés << " dólares" << endl;

  return 0;
}
```

4. Vuelva a escribir el siguiente programa de modo que la variable `voltios` sea cambiada a una constante simbólica.

```
#include <iostream>
using namespace std;

int main()
{
  double corriente, resistencia, voltios;

  voltios = 12;
  cout << " Introduzca la resistencia: ";
  cin >> resistencia;
  corriente = voltios / resistencia;
  cout << "La corriente es " << corriente << endl;

  return 0;
}
```

3.6 APLICACIONES

En esta sección se presentan dos aplicaciones para ilustrar más a fondo tanto el uso de la instrucción `cin` para aceptar datos introducidos por el usuario como el uso de la biblioteca de funciones para realizar cálculos.

Aplicación 1: Lluvia ácida

El uso de carbón como la fuente principal de energía por vapor comenzó con la Revolución Industrial. En la actualidad el carbón es una de las fuentes principales de generación de energía eléctrica en muchos países industrializados.

Desde mediados del siglo XIX se ha sabido que el oxígeno usado en el proceso de combustión se combina con el carbono y el azufre en el carbón para producir dióxido de carbono y dióxido de azufre. Cuando estos gases se liberan en la atmósfera el dióxido de azufre se combina con el agua y el oxígeno en el aire para formar ácido sulfúrico, el cual es transformado en iones hidronio y sulfatos separados (véase la figura 3.18). Son los iones hidronio en la atmósfera que caen a la tierra, como componentes de la lluvia, los que cambian los niveles de acidez de lagos y bosques.

El nivel de ácido de la lluvia y lagos se mide en una escala de pH usando la fórmula

$$\text{pH} = -\log_{10} (\text{concentración de iones hidronio})$$

donde la concentración de iones hidronio se mide en unidades de moles/litro. Un valor de pH de 7 indica un valor neutral (ni ácido ni alcalino), mientras niveles por debajo de 7 indican la presencia de un ácido, y niveles por encima de 7 indican la presencia de una sustancia alcalina. Por ejemplo, el ácido sulfúrico tiene un valor de pH de aproximadamente 1, la lejía tiene un valor de pH de aproximadamente 13, y el agua de manera típica tiene un valor de pH de 7. La vida marina por lo general no puede sobrevivir en agua con un nivel de pH por debajo de 4.

Usando la fórmula para el pH, se escribirá un programa en C++ que calcula el nivel de pH de una sustancia con base en un valor introducido por un usuario para la concentración de iones hidronio. Usando el procedimiento de desarrollo descrito en la sección 2.6 tenemos los siguientes pasos.

Paso 1 *Analizar el problema*

Aunque el planteamiento del problema proporciona información técnica sobre la composición de la lluvia ácida, desde un punto de vista de programación éste es un problema bastante simple. Aquí sólo se requiere una salida (un nivel de pH) y una entrada (la concentración de iones hidronio).

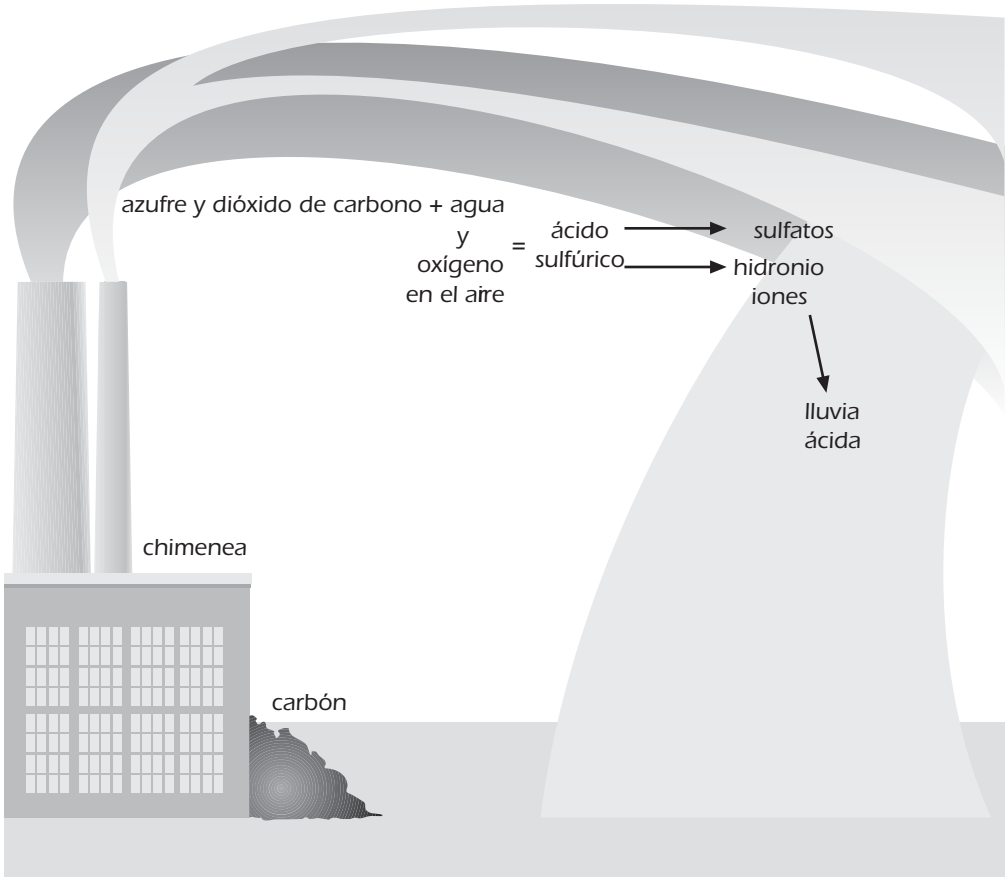


Figura 3.18 *La formación de lluvia ácida.*

Paso 2 Desarrollar una solución

El algoritmo requerido para transformar la entrada en la salida requerida es un uso bastante sencillo de la fórmula del pH que se proporciona. La representación en seudocódigo del algoritmo completo para introducir los datos de entrada, procesar los datos para producir la salida deseada y desplegar la salida es:

> ***Desplegar un indicador para introducir un nivel de concentración de iones.***
> ***Leer un valor para el nivel de concentración.***
> ***Calcular un nivel de pH usando la fórmula dada.***
> ***Desplegar el valor calculado.***

Para asegurar que entendemos la fórmula usada en el algoritmo, haremos un cálculo manual. El resultado de este cálculo puede usarse luego para verificar el resultado producido por el programa. Suponiendo una concentración de hidronio de 0.0001 (cualquier valor es útil), el nivel de pH se calcula como $-\log_{10} 10^{-4}$. Ya sea que sepa que el logaritmo de 10 elevado a una po-

tencia es la potencia misma, o usando una tabla de logaritmos, el valor de esta expresión es $-(-4) = 4$.

Paso 3 *Codificar la solución*

El programa 3.16 describe el algoritmo seleccionado en C++. La elección de los nombres de las variables es arbitrario.

Programa 3.16

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  double hidronio, nivelpH;

  cout << "Introduzca la concentración de iones hidronio: ";
  cin  >> hidronio;
  nivelpH = -log10(hidronio);
  cout << "El nivel de pH es " << nivelpH << endl;

  return 0;
}
```

El programa 3.16 comienza con dos instrucciones de preprocesador `#include`, seguidos por la función `main()`. Dentro de `main()`, una instrucción de declaración declara dos variables de punto flotante, `hidronio` y `nivelpH`. El programa despliega entonces un indicador solicitando datos de entrada del usuario. Después que se despliega el indicador, se usa una instrucción `cin` para almacenar los datos introducidos en la variable `hidronio`. Por último, se calcula un valor para `nivelpH`, usando la función logarítmica de biblioteca, y se despliega. Como siempre, el programa es terminado con una llave de cierre.

Paso 4 *Probar y corregir el programa*

Una ejecución de prueba del programa 3.16 produjo lo siguiente:

```
Introduzca el nivel de concentración de iones hidronio: 0.0001
El nivel de pH es 4
```

Debido a que el programa realiza un cálculo sencillo, y el resultado de esta ejecución de prueba concuerda con nuestro cálculo manual previo, el programa se ha probado por completo. Ahora puede usarse para calcular el nivel de pH de otras concentraciones de hidronio con confianza en que los resultados producidos son precisos.

Aplicación 2: Aproximación a la función exponencial

La función exponencial e^x, donde e se conoce como el número de Euler (y tiene el valor 2.718281828459045. . .) aparece muchas veces en descripciones de fenómenos naturales. Por ejemplo, la descomposición radiactiva, el crecimiento de la población y la curva normal (en forma de campana) usada en aplicaciones estadísticas pueden describirse usando esta función.

El valor de e^x puede aproximarse usando la serie[10]

$$1+\frac{x}{1}+\frac{x^2}{2}+\frac{x^3}{6}+\frac{x^4}{24}+\frac{x^5}{120}+\frac{x^6}{720}\cdots$$

Usando este polinomio como base, escriba un programa que aproxime e elevado a un valor de x introducido por un usuario utilizando los primeros cuatro términos de esta serie. Para cada aproximación despliegue el valor calculado por la función exponencial de C++, `exp()`, el valor aproximado y la diferencia absoluta entre los dos. Asegúrese de verificar su programa usando un cálculo manual. Una vez que esté completa la verificación, use el programa para aproximar e^4. Usando el procedimiento de desarrollo descrito en la sección 2.6 se llevan a cabo los siguientes pasos.

Paso 1 Analizar el problema

El planteamiento del problema especifica que se van a hacer cuatro aproximaciones, usando uno, dos, tres y cuatro términos del polinomio de aproximación, respectivamente. Para cada aproximación se requieren tres valores de salida: el valor de e^x producido por la función exponencial, el valor aproximado y la diferencia absoluta entre los dos valores. La figura 3.19 ilustra, en forma simbólica, la estructura del despliegue de salida requerido.

```
                e^x                    Aproximación       Diferencia
valor de la función en biblioteca  1er valor aproximado  1a diferencia
valor de la función en biblioteca   2o valor aproximado  2a diferencia
valor de la función en biblioteca  3er valor aproximado  3a diferencia
valor de la función en biblioteca   4o valor aproximado  4a diferencia
```

Figura 3.19 *Despliegue de salida requerido.*

La salida indicada en la figura 3.19 puede usarse para darse una "idea" de cómo se verá el programa. Al presentarse que cada línea en el despliegue sólo puede ser producida al ejecutar una instrucción `cout`, deberá quedar claro que deben ejecutarse cuatro de estas instrucciones. Además, en vista que cada línea de salida contiene tres valores calculados, cada instrucción `cout` tendrá tres elementos en su lista de expresión.

La única entrada al programa consiste en el valor de x. Por supuesto, esto requerirá un solo indicador y una instrucción `cin` para introducir el valor necesario.

[10]La fórmula de la que se deriva ésta es

$$e^x=\frac{x^0}{0!}+\frac{x^1}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}+\cdots+\frac{x^n}{n!}$$

Paso 2 Desarrollar una solución

Antes que puedan calcularse los elementos de salida, será necesario hacer que el programa le indique al usuario que introduzca un valor de x y luego haga que el programa acepte el valor introducido. El despliegue de salida real consiste de dos líneas seguidas por cuatro líneas de datos calculados. Las líneas de título pueden producirse usando dos instrucciones `cout`. Ahora veamos cómo se producen los datos reales que se están desplegando.

El primer elemento en la primera línea de salida de datos ilustrada en la figura 3.19 puede obtenerse usando la función de biblioteca `exp()`. El segundo elemento en esta línea, la aproximación a e^x, puede obtenerse usando el primer término en el polinomio que se dio en la especificación del programa. Por último, el tercer elemento en la línea puede calcularse usando la función de biblioteca `abs()` en la diferencia entre los primeros dos elementos. Cuando se calculan todos estos elementos, puede usarse una sola instrucción `cout` para desplegar los tres resultados en la misma línea.

La segunda línea de salida ilustrada en la figura 3.19 despliega el mismo tipo de elementos que la primera línea, excepto que la aproximación a e^x requiere el uso de dos términos del polinomio de aproximación. Hay que observar que además que el primer elemento en la segunda línea, el valor obtenido por la función `exp()`, es el mismo que el primer elemento en la primera línea. Esto significa que este elemento no tiene que recalcularse y tan sólo puede desplegarse una segunda vez el valor calculado para la primera línea. Una vez que se han calculado los datos para la segunda línea, puede usarse una sola instrucción `cout` para desplegar los valores requeridos.

Por último, sólo el segundo y tercer elementos en las últimas dos líneas de salida mostradas en la figura 3.19 necesitan recalcularse, en vista que el primer elemento en estas líneas es el mismo que se calculó antes para la primera línea.

Por tanto, para este problema, el algoritmo completo descrito en seudocódigo es

Desplegar un indicador para el valor de entrada de x
Leer el valor de entrada
Desplegar las líneas de encabezado
Calcular el valor exponencial de x usando la función exp()
Calcular la primera aproximación
Calcular la primera diferencia
Imprimir la primera línea de salida
Calcular la segunda aproximación
Calcular la segunda diferencia
Imprimir la segunda línea de salida
Calcular la tercera aproximación
Calcular la tercera diferencia
Imprimir la tercera línea de salida
Calcular la cuarta aproximación
Calcular la cuarta diferencia
Imprimir la cuarta línea de salida

Para asegurar que entendemos el procesamiento usado en el algoritmo se hará un cálculo manual. El resultado de este cálculo puede utilizarse luego para verificar el resultado producido por el programa que escribimos. Para propósitos de prueba se usará un valor de 2 para x, el cual produce las siguientes aproximaciones.

Usando el primer término del polinomio la aproximación es

$$e^2 = 1$$

Usando los primeros dos términos del polinomio la aproximación es

$$e^2 = 1 + \frac{2}{1} = 3$$

Usando los primeros tres términos del polinomio la aproximación es

$$e^2 = 3 + \frac{2^2}{2} = 5$$

Usando los primeros cuatro términos del polinomio la aproximación es

$$e^2 = 5 + \frac{2^3}{6} = 6.3333$$

Hay que observar que al usar cuatro términos del polinomio no fue necesario recalcular el valor de los primeros tres términos; en cambio, se usó el valor calculado antes.

Paso 3 Codificar la solución

El programa 3.17 representa una descripción del algoritmo seleccionado en C++.

Programa 3.17

```
// este programa aproxima la funcion e elevada a la potencia x
// usando uno, dos, tres y cuatro terminos de un polinomio de aproximacion
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  double x, val_func, aprox, diferencia;

  cout << "\nIntroduzca un valor de x: ";
  cin  >> x;

    // imprimir las dos lineas del titulo
  cout << " e a la x          Aproximacion       Diferencia\n";
  cout << "-------------     -------------      -------------\n";

  val_func = exp(x);      // utilizar la funcion de biblioteca

    // calcular la primera aproximacion
```

(*continúa*)

(Continuación)

```
    aprox = 1;
    diferencia = abs(val_func - aprox);
    cout << setw(10) << setiosflags(ios::showpoint) << val_func
         << setw(18) << aprox
         << setw(18) << diferencia << endl;

      // calcular la segunda aproximacion
    aprox = aprox + x;
    diferencia = abs(val_func - aprox);
    cout << setw(10) << setiosflags(ios::showpoint) << val_func
         << setw(18) << aprox
         << setw(18) << diferencia << endl;

      // calcular la tercera aproximacion
    aprox = aprox + pow(x,2)/2.0;
    diferencia = abs(val_func - aprox);
    cout << setw(10) << setiosflags(ios::showpoint) << val_func
         << setw(18) << aprox
         << setw(18) << diferencia << endl;

      // calcular la cuarta aproximacion
    aprox = aprox + pow(x,3)/6.0;
    diferencia = abs(val_func - aprox);
    cout << setw(10) << setiosflags(ios::showpoint) << val_func
         << setw(18) << aprox
         << setw(18) << diferencia << endl;

    return 0;
}
```

Al revisar el programa 3.17 se puede observar que el valor de entrada de x se obtiene primero. Luego se imprimen las dos líneas del título antes que se haga cualquier cálculo. Luego se calcula el valor de e^x usando la función de biblioteca `exp()` y se asigna a la variable `val_func`. Esta asignación permite que este valor sea usado en los cuatro cálculos de diferencias y se despliegue cuatro veces sin necesidad de recalcularse.

En vista que la aproximación a e^x se "acumula" usando cada vez más términos del polinomio de aproximación, sólo se calcula el término nuevo para cada aproximación y se agrega a la aproximación previa. Por último, para permitir que las mismas variables se usen de nuevo, los valores en ellas se imprimen de inmediato antes que se haga la siguiente aproximación. La siguiente es una muestra de ejecución producida por el programa 3.17.

```
Introduzca un valor de x: 2
 e a la x          Aproximación        Diferencia
-------------     -------------     -------------
  7.38906           1.00000           6.38906
  7.38906           3.00000           4.38906
  7.38906           5.00000           2.38906
  7.38906           6.33333           1.05572
```

Paso 4 Probar y corregir el programa

Las primeras dos columnas de datos de salida producidos por la muestra de ejecución concuerdan con nuestro cálculo manual. Una comprobación manual de la última columna verifica que también contiene en forma correcta la diferencia en valores entre las primeras dos columnas.

Debido a que el programa sólo ejecuta nueve cálculos, y el resultado de la ejecución de prueba concuerda con nuestros cálculos manuales, parece que el programa se ha probado por completo. Sin embargo, es importante entender que esto se debe a nuestra elección de los datos de prueba. Seleccionar un valor de 2 para x nos obligó a verificar que el programa estaba, de hecho, calculando 2 elevado a las potencias requeridas. Una elección de 0 o 1 para nuestro cálculo manual no nos habría proporcionado la verificación que necesitamos. ¿Puede ver por qué sucede así?

Usar estos últimos dos valores no probaría de manera adecuada si el programa usó la función `pow()` en forma adecuada, ¡o ni siquiera si la usó en absoluto! Es decir, podría haberse construido un programa incorrecto que no usara la función `pow()` para producir valores correctos para = 0 y x = 1, pero no para otros valores de x. Sin embargo, en vista que los datos de prueba que usamos verifican de manera adecuada el programa, podemos usarlo con confianza en los resultados producidos. Es claro, sin embargo, que la salida demuestra que para lograr cualquier nivel de precisión con el programa se requerirían más de cuatro términos.

Ejercicios 3.6

1. Introduzca, compile y ejecute el programa 3.16 en su computadora.
2. **a.** Introduzca, compile y ejecute el programa 3.17 en su computadora.
   b. Determine cuántos términos del polinomio de aproximación deberían usarse para lograr un error de menos de 0.0001 entre la aproximación y el valor de e^2 determinado por la función `exp()`.
3. Por error un estudiante escribió el programa 3.17 como sigue:

```
// este programa aproxima la funcion e elevada a la potencia x
// usando uno, dos, tres y cuatro terminos de un polinomio de aproximacion
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  double x, Val_func, aprox, diferencia;
```

```
    // imprimir las dos lineas del titulo
  cout << " e  a  la x       Aproximacion       Diferencia\n";
  cout << "-------------     -------------      -------------\n";

  cout << "\nIntroduzca un valor de x: ";
  cin >> x;
  val_func = exp(x);      // usar la funcion de biblioteca

    // calcular la primera aproximacion
  aprox = 1;
  diferencia = abs(val_func - aprox);
  cout << setw(10) << setiosflags(ios::showpoint) << val_func
       << setw(18) << aprox
       << setw(18) << diferencia << endl;

    // calcular la segunda aproximacion
  aprox = aprox + x;
  diferencia = abs(val_func - aprox);
  cout << setw(10) << setiosflags(ios::showpoint) << val_func
       << setw(18) << aprox
       << setw(18) << diferencia << endl;

    // calcular la tercera aproximacion
  aprox = aprox + pow(x,2)/2.0;
  diferencia = abs(val_func - aprox);
  cout << setw(10) << setiosflags(ios::showpoint) << val_func
       << setw(18) << aprox
       << setw(18) << diferencia << endl;

    // calcular la cuarta aproximacion
  aprox = aprox + pow(x,3)/6.0;
  diferencia = abs(val_func - aprox);
  cout << setw(10) << setiosflags(ios::showpoint) << val_func
       << setw(18) << aprox
       << setw(18) << diferencia << endl;

  return 0;
}
```

Determine la salida que producirá este programa.

4. El valor de π puede aproximarse con la serie

$$4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots\right)$$

Usando esta fórmula, escriba un programa que calcule y despliegue el valor de π usando 2, 3 y 4 términos de la serie.

5. a. La fórmula para la desviación normal estándar, z, usada en aplicaciones estadísticas es

$$z = \frac{x - \mu}{\sigma}$$

donde μ se refiere a un valor medio y σ a una desviación estándar. Usando esta fórmula, escriba un programa que calcule y despliegue el valor de la desviación normal estándar cuando $x = 85.3$, $\mu = 80$ y $\sigma = 4$.

b. Vuelva a escribir el programa elaborado en el ejercicio 5a para aceptar los valores de x, μ y σ como entradas del usuario mientras se está ejecutando el programa.

6. a. La ecuación de la curva normal (en forma de campana) utilizada en aplicaciones estadísticas es

$$y = \frac{1}{\sigma\sqrt{2\pi}} e^{-\left(\frac{1}{2}\right)\left[(x-\mu)/\sigma\right]^2}$$

Usando esta ecuación, y suponiendo que $\mu = 90$ y $\sigma = 4$, escriba un programa que determine y despliegue el valor de y cuando $x = 80$.

b. Vuelva a escribir el programa elaborado en el ejercicio 6a para aceptar los valores de x, μ y σ como entradas del usuario mientras se está ejecutando el programa.

7. a. Escriba, compile y ejecute un programa en C++ que calcule y despliegue el aumento de voltaje de un amplificador de tres etapas a una frecuencia de 1000 Hertz. Los aumentos de voltaje de las etapas son:

Aumento de la etapa 1: $23/[2.3^2 + (0.044f)^2]^{1/2}$
Aumento de la etapa 2: $12/[6.7^2 + (0.34f)^2]^{1/2}$
Aumento de la etapa 3: $17/[1.9^2 + (0.45f)^2]^{1/2}$

donde f es la frecuencia en Hertz. El aumento de voltaje del amplificador es el producto de los aumentos de las etapas individuales.

b. Vuelva a hacer el ejercicio 7a suponiendo que la frecuencia se introducirá cuando el programe esté en ejecución.

8. El volumen de petróleo almacenado en un tanque cilíndrico enterrado en el subsuelo a 200 pies de profundidad está determinado por la medición de la distancia de la parte superior del tanque a la superficie del petróleo. Conociendo esta distancia y el radio del tanque, el volumen de petróleo en el tanque puede determinarse usando la fórmula *volumen* = π *radio*^{2} (200 – *distancia*). Usando esta información, escriba, compile y ejecute un programa en C++ que acepte las mediciones del radio y la distancia, calcule el volumen de petróleo en el tanque y despliegue los dos valores de entrada y el volumen calculado. Verifique los resultados de su programa haciendo un cálculo manual usando los siguientes datos de prueba: radio igual a 10 pies y distancia igual a 12 pies.

9. El perímetro, el área de superficie aproximada y el volumen aproximado de una alberca están dados por las siguientes fórmulas:

perímetro = 2(largo + ancho)
*volumen = largo * ancho * profundidad promedio*
*área de superficie subterránea = 2(largo + ancho)profundidad promedio + largo * ancho*

Usando estas fórmulas como base, escriba un programa en C++ que acepte las medidas de largo, ancho y profundidad promedio, y luego calcule el perímetro, el volumen y el área de superficie subterránea de la alberca. Al escribir su programa haga los siguientes dos cálculos inmediatamente después que se han introducido los datos de entrada: *largo* * *ancho* y *largo* + *ancho*. Los resultados de estos dos cálculos deberán usarse luego, según sea apropiado, en las instrucciones de asignación para determinar el perímetro, volumen y área de superficie subterránea sin recalcularlas para cada ecuación. Verifique los resultados de su programa haciendo un cálculo manual usando los siguientes datos de prueba: largo igual a 25 pies, ancho igual a 15 pies y profundidad promedio igual a 5.5 pies. Cuando haya verificado que su programa funciona, úselo para completar la siguiente tabla.

Largo	Ancho	Profundidad	Perímetro	Volumen	Área de superficie subterránea
25	10	5.0			
25	10	5.5			
25	10	6.0			
25	10	6.5			
30	12	5.0			
30	12	5.5			
30	12	6.0			
30	12	6.5			

3.7 Errores comunes de programación

Al usar el material presentado en este capítulo, esté consciente de los siguientes errores posibles:

1. Olvidar asignar o inicializar valores para todas las variables antes que éstas se usen en una expresión. Tales valores pueden ser asignados por instrucciones de asignación, ser inicializados dentro de una instrucción de declaración o asignados en forma interactiva introduciendo los valores usando el objeto `cin`.
2. Utilizar una función matemática de biblioteca sin incluir la declaración de preprocesador `#include <cmath>` (y en un sistema basado en UNIX olvidar incluir el argumento `-lm` en la línea de comandos `cc`).
3. Utilizar una función de biblioteca sin proporcionar el número correcto de argumentos que tengan el tipo de datos apropiado.
4. Aplicar el operador de incremento o decremento a una expresión. Por ejemplo, la expresión

   ```
   (count + n)++
   ```

es incorrecta. Los operadores de incremento y decremento sólo pueden aplicarse a variables individuales.

5. Olvidar separar todas las variables transmitidas a `cin` con un símbolo de extracción, `>>`.
6. No estar dispuesto a probar un programa a fondo. Después de todo, en vista que usted escribió el programa, supone que es correcto o lo habría cambiado antes de compilarlo. Es difícil en extremo retroceder y probar con honestidad su propio software. Como programador deberá recordar en forma constante que un programa no es correcto por el solo hecho que usted piense que lo es. Encontrar errores en su propio programa es una experiencia seria, pero le ayudará a convertirse en un programador maestro.
7. Un error más exótico y menos común ocurre cuando se usan los operadores de incremento y decremento con variables que aparecen más de una vez en la misma expresión. Este error ocurre debido a que C++ no especifica el orden en el que se tiene acceso a los operandos dentro de una expresión. Por ejemplo, el valor asignado a resultado en la instrucción

   ```
   resultado = i + i++;
   ```

   es dependiente del compilador. Si su compilador tiene acceso primero al primer operando, `i`, la instrucción anterior es equivalente a

   ```
   result = 2 * i;
   i++;
   ```

   Sin embargo, si su compilador tiene acceso primero al segundo operando, `i++`, el valor del primer operando será alterado antes que se use la segunda vez y el valor $2i + 1$ es asignado al resultado. Por consiguiente, como regla general, no use el operador de incremento o decremento en una expresión cuando la variable sobre la que opera aparece más de una vez en la expresión.

3.8 RESUMEN DEL CAPÍTULO

1. Una expresión es una secuencia de uno o más operandos separados por operadores. Un operando es una constante, una variable u otra expresión. Un valor se asocia con una expresión.
2. Las expresiones se evalúan de acuerdo con la precedencia y asociatividad de los operadores usados en la expresión.
3. El símbolo de asignación, =, es un operador. Las expresiones que usa este operador asignan un valor a una variable; además, la expresión en sí adquiere un valor. En vista que la asignación es una operación en C++, son posibles múltiples usos del operador de asignación en la misma expresión.

4. El operador de incremento, `++`, agrega uno a una variable, mientras el operador de decremento, `--`, resta uno de una variable. Ambos operadores pueden ser usados como prefijos o posfijos. En la operación de prefijo la variable es aumentada (o disminuida) antes que su valor sea usado. En la operación de posfijo la variable es aumentada (o disminuida) después que se usa su valor.

5. C++ proporciona funciones de biblioteca para calcular raíz cuadrada, logaritmos y otros cálculos matemáticos. Cada programa que utilice una de estas funciones matemáticas debe incluir la instrucción `#include <cmath>` o tener una declaración de función para la función matemática antes de llamarla.

6. Todas las funciones matemáticas de biblioteca operan sobre sus argumentos para calcular un solo valor. Para usar una función de biblioteca de manera efectiva, debe saber lo que hace la función, el nombre de la función, el número y tipos de datos de los argumentos esperados por la función y el tipo de datos del valor devuelto.

7. Los datos transmitidos a una función se llaman argumentos de la función. Los argumentos son transmitidos a una función de biblioteca al incluir cada argumento, separado por comas, dentro de los paréntesis que siguen al nombre de la función. Cada función tiene sus propios requisitos para el número y tipos de datos de los argumentos que deben proporcionarse.

8. Las funciones pueden incluirse dentro de expresiones más grandes.

9. El objeto `cin` se usa para introducir datos. Este objeto acepta un flujo de datos del teclado y asigna los datos a variables. La forma general de una instrucción que utiliza `cin` es:

   ```
   cin >> var1 >> var2 . . . >> varn;
   ```

   El símbolo de extracción, `>>`, debe usarse para separar los nombres de las variables.

10. Cuando encuentra una instrucción `cin` la computadora suspende de manera temporal la ejecución de más instrucciones hasta que se hayan introducido suficientes datos para el número de variables contenidas en la instrucción `cin`.

11. Es una buena práctica de programación desplegar un mensaje, antes de una instrucción `cin`, que alerte al usuario sobre el tipo y número de elementos de datos que deben introducirse. Dicho mensaje se llama indicador.

12. Los valores pueden equipararse a una sola constante, usando la palabra clave `const`. Esto crea una constante nombrada que es de sólo lectura después que es inicializada dentro de la instrucción de declaración. Esta declaración tiene la sintaxis

    `const` *Tipodedatos NombreSimbólico* = *valorInicial;*

    y permite que se use la constante en lugar del valor inicial en cualquier parte del programa después de la declaración.

3.9 UN ACERCAMIENTO MÁS A FONDO: ERRORES DE PROGRAMACIÓN

El ideal en la programación es producir programas legibles libres de errores que funcionen en forma correcta y puedan modificarse o cambiarse con un mínimo de pruebas. Puede trabajar hacia este ideal teniendo en cuenta los diferentes tipos de errores que pueden ocurrir, cuándo se detectan de manera típica y cómo corregirlos.

Puede detectar un error en cuatro formas:

1. Antes que un programa sea compilado
2. Mientras el programa se compila
3. Mientras el programa se ejecuta
4. Después que el programa se ha ejecutado y se ha examinado la salida

Y, por extraño que parezca, en algunos casos, un error puede no detectarse en absoluto.

El método para detectar errores antes que se compile un programa se llama **verificación de escritorio**. La verificación de escritorio, la cual por lo general se lleva a cabo mientras se encuentra sentado ante un escritorio con el código enfrente de usted, se refiere al proceso de verificar el código fuente en busca de errores inmediatamente después que ha sido mecanografiado.

Los errores detectados por el compilador se conocen de manera formal como **errores en tiempo de compilación**, y los errores que ocurren mientras el programa se ejecuta se conocen de manera formal como errores en tiempo de ejecución. Otros nombres para los errores en tiempo de compilación son **errores** de **sintaxis** y **errores** de **análisis** gramatical, términos que enfatizan el tipo de error que es detectado por el compilador.

En este momento, es probable que haya encontrado numerosos errores en tiempo de compilación. Aunque los programadores principiantes tienden a frustrarse por ellos, los programadores experimentados entienden que el compilador está realizando una verificación valiosa, y que corregir los errores que detecte el compilador por lo general es fácil. Debido a que estos errores ocurren mientras se está desarrollando el programa y no mientras un usuario intenta realizar una tarea importante, nadie excepto el programador sabe que ocurrieron; los arregla y se van.

Los errores en tiempo de ejecución son más problemáticos debido a que ocurren mientras un usuario ejecuta el programa; en la mayor parte de los sistemas comerciales, el usuario no es el programador. Aunque muchos tipos de errores pueden causar un error en tiempo de ejecución, como una falla en el hardware, desde un punto de vista de programación la mayor parte de los errores en tiempo de ejecución se conocen como errores de lógica o lógica defectuosa, lo cual abarca no haber pensado lo que el programa debería hacer o no anticipar cómo un usuario puede hacer que falle el programa. Por ejemplo, si un usuario introduce datos que producen un intento de dividir un número entre cero, ocurre un error en tiempo de ejecución. Como programador, la única forma de protegerse contra errores en tiempo de ejecución es anticipar todo lo que podría hacer una persona para causar errores y someter su programa a una prueba rigurosa. Aunque los programadores principiantes tienden a culpar al usuario por un error causado al introducir datos incorrectos, los profesionales no lo hacen. Entienden que un error en tiempo de ejecución es un defecto en el producto final que puede causar daños a la reputación del programa y el programador.

Para prevenir errores en tiempo de compilación y en tiempo de ejecución, es más fructífero distinguir entre ellos basándose en lo que los causa. Como se ha señalado, los errores de

compilación también se llaman errores de sintaxis, lo cual se refiere a errores en la estructura u ortografía de una instrucción. Por ejemplo, examine las siguientes instrucciones:

```
cout << "Hay cuatro errores de sintaxis aqui\n
cot " Puede encontralos";
```

Contienen cuatro errores de sintaxis. Estos errores son los siguientes:

1. Faltan las comillas que cierran en la línea 1.
2. Falta un punto y coma para terminar en la línea 1.
3. La palabra clave `cout` está mal escrita en la línea 2.
4. Falta el símbolo de inserción, <<, en la línea 2.

Todos estos errores serán detectados por el compilador cuando el programa es compilado. Esto sucede con todos los errores de sintaxis porque violan las reglas básicas de C++; si no son descubiertos por la verificación de escritorio, el compilador los detecta y despliega un mensaje de error.[11] En algunos casos, el mensaje de error es claro y el error es obvio; en otros casos, se requiere un poco de trabajo detectivesco para entender el mensaje de error desplegado por el compilador. Debido a que los errores de sintaxis son el único tipo de error que puede detectarse en el momento de la compilación, los términos errores en tiempo de compilación y errores de sintaxis se usan de manera indistinta. En sentido estricto, sin embargo, tiempo de compilación se refiere al momento en que se detecta el error y sintaxis se refiere al tipo de error detectado.

El error en la palabra "encontralos" en la segunda instrucción no es un error de sintaxis. Aunque este error de ortografía producirá que se despliegue una línea de salida indeseable, no es una violación de las reglas sintácticas de C++. Es un **error tipográfico**, conocido por lo común como "error de dedo".

Un error lógico puede causar un error en tiempo de ejecución o producir resultados incorrectos. Estos errores se caracterizan por una salida errónea, inesperada o involuntaria que es un resultado directo de algún defecto en la lógica del programa. Estos errores, los cuales nunca son detectados por el compilador, pueden detectarse en la verificación de escritorio, al probar el programa, por accidente cuando un usuario obtiene una salida errónea mientras el programa se está ejecutando, o no detectarse en absoluto. Si el error es detectado mientras el programa está en ejecución, puede ocurrir un error en tiempo de ejecución que produce que se genere un mensaje de error, la terminación prematura del programa, o ambos.

El error de lógica más grave es causado por una comprensión incorrecta de los requerimientos totales del programa, debido a que la lógica dentro de un programa se refleja en la lógica con la que es codificado. Por ejemplo, si el propósito de un programa es calcular la fuerza de soporte de carga de una viga de acero y el programador no entiende por completo cómo se va a hacer el cálculo, qué entradas son necesarias para realizar el cálculo o qué condiciones especiales existen (como la forma en que la temperatura afecta a la viga), ocurrirá un error de lógica. Debido a que estos errores no son detectados por el compilador y con frecuencia pueden pasarse por alto en el tiempo de ejecución, siempre son más difíciles de detectar que los errores de sintaxis. Si son detectados, un error de lógica de manera típica aparece en una de dos formas predominantes. En un caso, el programa se ejecuta has-

[11] Sin embargo, puede ser que no se detecten todos al mismo tiempo. Con frecuencia, un error de sintaxis enmascara a otro error, y el segundo error es detectado después que se corrige el primer error.

ta completarse pero produce resultados incorrectos. Por lo general, los errores de lógica de este tipo son revelados por lo siguiente:

- **No hay salida.** Esto es causado por una omisión en una instrucción de salida o una secuencia de instrucciones que elude de manera inadvertida una instrucción de salida.
- **Salida poco atractiva o mal alineada.** Esto es causado por un error en una instrucción de salida.
- **Resultados numéricos incorrectos.** Esto es causado por valores incorrectos asignados a las variables usadas en una expresión, el uso de una expresión aritmética incorrecta, una omisión de una instrucción, un error de redondeo o el uso de una secuencia de instrucciones inapropiada.

Una segunda forma en que se revelan los errores de lógica es causando un error en tiempo de ejecución. Son ejemplos de este tipo de error de lógica son los intentos de dividir entre cero u obtener la raíz cuadrada de un número negativo.

Deberá planear la prueba de su programa cuidadosamente para maximizar la posibilidad de localizar errores. Siempre tenga en cuenta que *aunque una sola prueba puede revelar la presencia de un error, no verifica la ausencia de otro error*. Es decir, el hecho que un error sea revelado por la prueba, no indica que otro error no esté al acecho en alguna otra parte en el programa; además, *el hecho que una prueba no revele errores no significa que no haya errores*.

Sin embargo, una vez que descubre un error debe localizar dónde ocurre y arreglarlo. En jerga de computación, un error de programa se conoce como **bug**, y el proceso de aislar, corregir y verificar la corrección se llama **depuración**.[12]

Aunque no existen reglas inflexibles para aislar la causa de un error, pueden aplicarse algunas técnicas útiles. La primera de éstas es una técnica preventiva. Con frecuencia, muchos errores son introducidos por el programador por la premura de codificar y ejecutar un programa antes de entender qué se requiere y cómo se va a lograr el resultado. Un síntoma de esta prisa por introducir un programa en la computadora es la falta de un esbozo del programa propuesto o la falta de una comprensión detallada de lo que se requiere en realidad. Muchos errores pueden eliminarse al verificar en el escritorio una copia del programa antes de introducirlo o compilarlo.

Una segunda técnica útil es imitar a la computadora y ejecutar cada instrucción en forma manual, como lo haría la computadora. Esto significa escribir cada variable, tal como se encuentra en el programa, y enumerar el valor que debería almacenarse en la variable conforme se encuentre cada entrada e instrucción de asignación. Hacer esto agudiza sus habilidades de programación porque requiere que entienda lo que causa que suceda cada instrucción en su programa. Esta verificación se llama **rastreo del programa**.

Una tercera técnica de depuración poderosa es incluir algún código temporal en su programa que despliegue los valores de variables selectas. Si los valores desplegados son incorrectos, puede determinar qué parte de su programa los generó y hacer las correcciones necesarias.

[12]La derivación de este término es interesante. Cuando un programa dejó de ejecutarse en la computadora MARK I, en la Universidad de Harvard, en septiembre de 1945, el mal funcionamiento fue rastreado hasta un insecto muerto que había entrado en los circuitos eléctricos. La programadora, Grace Hopper, registró el incidente en su bitácora como "Primer caso real de bug (insecto) encontrado".

En la misma manera, podría agregar código temporal que despliegue los valores de todos los datos de entrada. Esta técnica se conoce como **impresión en eco** y es útil para establecer que el programa está recibiendo en forma correcta e interpretando en forma correcta los datos de entrada.

La más poderosa de todas las técnicas de depuración y rastreo es usar un programa especial llamado **depurador**. Un programa depurador puede controlar la ejecución de un programa en C++, puede interrumpir el programa C++ en cualquier punto de su ejecución y desplegar los valores de todas las variables en el punto de interrupción.

Por último, ninguna exposición de la depuración está completa sin mencionar el ingrediente primario necesario para el aislamiento y corrección exitosa de los errores. Es la actitud y espíritu con que se emprende la tarea. Después de escribir un programa, es natural que suponga que es correcto. Es difícil retroceder y probar honestamente y encontrar errores en su propio software. Como programador, debe recordar en forma constante que un programa no es correcto sólo porque usted piensa que lo es. Encontrar errores en su propio programa es una experiencia seria, pero le ayudará a que se convierta en un programador maestro. El proceso puede ser emocionante y divertido si lo enfoca como una detección de problemas con usted como el detective maestro.

Consideración de opciones de carrera

Ingeniería mecánica

En general, los ingenieros mecánicos trabajan con máquinas o sistemas que producen o aplican energía. El rango de actividades tecnológicas que se consideran parte de la ingeniería mecánica quizás es más amplio que en cualquier otro campo de la ingeniería. El campo puede subdividirse más o menos en cuatro categorías.

1. Energía. Diseño de máquinas y sistemas generadores de energía como quemadores y turbinas para generar electricidad, energía solar, sistemas de calefacción e intercambio de calor.
2. Diseño. Diseño innovador de partes o componentes de máquinas desde los más intrincados y pequeños hasta los gigantescos. Por ejemplo, los ingenieros mecánicos trabajan al lado de los ingenieros eléctricos para diseñar sistemas de control automático como los robots.
3. Automotriz. Diseño y prueba de vehículos de transporte y las máquinas usadas para fabricarlos.
4. Calefacción, ventilación, aire acondicionado y refrigeración. Diseño de sistemas para controlar nuestro ambiente tanto en interiores como en exteriores y para controlar la contaminación.

Los ingenieros mecánicos por lo general tienen estudios sólidos en materias como termodinámica, transferencia de calor, estática y dinámica y mecánica de fluidos.

CAPÍTULO 4
Estructuras de selección

TEMAS

4.1 CRITERIOS DE SELECCIÓN
OPERADORES LÓGICOS
UN PROBLEMA DE EXACTITUD NUMÉRICA

4.2 LA INSTRUCCIÓN `if-else`
INSTRUCCIONES COMPUESTAS
EL ALCANCE DE UN BLOQUE
SELECCIÓN UNIDIRECCIONAL
PROBLEMAS ASOCIADOS CON LA INSTRUCCIÓN `if-else`

4.3 INSTRUCCIONES `if` ANIDADAS
LA CADENA `if-else`

4.4 LA INSTRUCCIÓN `switch`

4.5 APLICACIONES
APLICACIÓN 1: VALIDACIÓN DE DATOS
APLICACIÓN 2: RESOLVER ECUACIONES CUADRÁTICAS

4.6 ERRORES COMUNES DE PROGRAMACIÓN

4.7 RESUMEN DEL CAPÍTULO

4.8 APÉNDICE DEL CAPÍTULO: UN ACERCAMIENTO MÁS A FONDO A LA PRUEBA EN PROGRAMACIÓN
CONSIDERACIÓN DE LAS OPCIONES DE CARRERA: INGENIERÍA CIVIL

Han ocurrido muchos avances en los fundamentos teóricos de la programación desde el comienzo de los lenguajes de alto nivel a finales de la década de los años 50. Uno de los más importantes de estos avances fue el reconocimiento a finales de la década de los años 60 que cualquier algoritmo, sin importar cuán complejo fuera, podía ser construido usando combinaciones de cuatro estructuras de ***flujo de control*** *estandarizadas: secuencial, de selección, de repetición y de invocación.*

El término ***flujo de control*** *se refiere al orden en que las instrucciones de un programa son ejecutadas. A menos que se dirijan de otra manera, el flujo de control normal para todos los programas es* ***secuencial****. Esto significa que las instrucciones son ejecutadas en secuencia, una tras otra, en el orden en que son colocadas dentro del programa.*

Las estructuras de selección, repetición e invocación permiten que el flujo de control secuencial sea alterado en formas definidas con precisión. Como podrá haber adivinado, la estructura de selección se usa para seleccionar cuáles instrucciones se han de ejecutar a continuación y la estructura de repetición se usa para repetir un conjunto de instrucciones. En este capítulo presentamos las instrucciones de selección de C++. Las técnicas de repetición e invocación se presentan en los capítulos 5 y 6.

4.1 CRITERIOS DE SELECCIÓN

En la solución de muchos problemas, deben emprenderse diferentes acciones dependiendo del valor de los datos. Los ejemplos de situaciones simples incluyen calcular un área *sólo si* las mediciones son positivas, ejecutar una división *sólo si* el divisor no es cero, imprimir diferentes mensajes *dependiendo* del valor de una calificación recibida, y así en forma sucesiva.

La instrucción `if-else` en C++ se usa para poner en práctica una estructura de decisión en su forma más simple, la de elegir entre dos alternativas. La sintaxis de seudocódigo más usada de esta instrucción es

```
if (condición)
  instrucción ejecutada si la condición es verdadera;
else
  instrucción ejecutada si la condición es falsa;
```

Cuando un programa en ejecución encuentra la instrucción `if`, la condición es evaluada para determinar su valor numérico, el cual es interpretado entonces como verdadero o falso. Si la condición produce cualquier valor numérico positivo o negativo diferente de cero, la condición es considerada como una condición "verdadera" y se ejecuta la instrucción que sigue a `if`. Si la condición produce un valor numérico de cero, la condición es considerada como una condición "falsa" y se ejecuta la instrucción que sigue a `else`. La parte `else` de la instrucción es opcional y puede omitirse.

La condición usada en una instrucción `if` puede ser cualquier expresión válida de C++ (incluyendo, como se verá, una expresión de asignación). Las expresiones más usadas por lo común, sin embargo, se llaman **expresiones relacionales**. Una **expresión relacional simple** consiste en un operador relacional que compara dos operandos, como se muestra en la figura 4.1.

Figura 4.1 Anatomía de una expresión relacional simple.

Mientras cada operando en una expresión relacional puede ser una variable o una constante, los operadores relacionales deben ser uno de los expuestos en la tabla 4.1. Estos operadores relacionales pueden usarse con operados de números enteros, de punto flotante, de precisión doble o de carácter, pero deben escribirse con exactitud como se muestra en la tabla 4.1. Por tanto, mientras todos los siguientes ejemplos son válidos:

```
edad > 40     largo <= 50              temp > 98.6
   3 < 4      indicador == terminar    Num_id == 682
dia != 5      2.0 > 3.3                horas > 40
```

los siguientes son inválidos:

```
largo =< 50             // operador fuera de orden
2.0 >> 3.3              // operador inválido
indicador = = terminar // no se permiten espacios
```

Tabla 4.1

Operador relacional	Significado	Ejemplo
<	menor que	`edad < 30`
>	mayor que	`altura > 6.2`
<=	menor que o igual a	`gravable <= 20000`
>=	mayor que o igual a	`temp >= 98.6`
==	igual a	`calificación == 100`
!=	no es igual a	`número != 250`

Las expresiones relacionales a veces se llaman **condiciones** y se usarán ambos términos para referirnos a ellas. Como todas las expresiones C++, las expresiones relacionales son evaluadas para producir un resultado numérico.[1] En el caso de una expresión relacional, el valor de la expresión sólo puede ser el valor entero de 1 o 0, el cual es interpretado como verdadero y falso, respectivamente. *Una expresión relacional que interpretaríamos como verdadera produce un valor entero de 1, y una expresión relacional falsa produce un valor entero de 0*. Por ejemplo, debido a que la relación 3 < 4 siempre es verdadera, esta expresión tiene un valor de 1, y debido a que la relación 2.0 > 3.3 siempre es falsa, el valor de la expresión en sí es 0. Esto puede verificarse usando las instrucciones

```
cout << "El valor de 3 < 4 es " << (3 < 4) << endl;
cout << "El valor de 2.0 > 3.0 es " << (2.0 > 3.3) << endl;
cout << "El valor de verdadero es " << verdadero << endl;
cout << "El valor de falso es " << falso << endl;
```

[1]En este aspecto C++ difiere de otros lenguajes de computadora de nivel alto que producen un resultado booleano (verdadero, falso).

lo cual produce el despliegue

```
El valor de 3 < 4 es 1
El valor de 2.0 > 3.0 es 0
El valor de verdadero es 1
El valor de falso es 0
```

El valor de una expresión relacional como `horas > 40` depende del valor almacenado en la variable `horas`.

En un programa C++, el valor de una expresión relacional no es tan importante como la interpretación que C++ coloca en el valor cuando se usa la expresión como parte de una instrucción de selección. En estas instrucciones, las cuales se presentan en la siguiente sección, se verá que un valor de cero es usado por C++ para representar una condición falsa y cualquier valor diferente de cero se utiliza para representar una condición verdadera. La selección de cuál instrucción ejecutar a continuación se basa entonces en el valor obtenido.

Además de los operandos numéricos, pueden compararse datos de carácter usando operadores relacionales. Para estas comparaciones, los valores `char` son coaccionados de manera automática a valores `int` para la comparación. Por ejemplo, en el código Unicode, la letra 'A' se almacena usando un código que tiene un valor numérico inferior que la letra 'B', el código para 'B' tiene un valor inferior que el código para 'C', y así en forma sucesiva. Para conjuntos de caracteres codificados de esta manera, las siguientes condiciones se evalúan como sigue:

Expresión	Valor	Interpretación
`'A' > 'C'`	0	falso
`'D' <= 'Z'`	1	verdadero
`'E' == 'F'`	0	falso
`'g' >= 'm'`	0	falso
`'b' != 'c'`	1	verdadero
`'a' == 'A'`	0	falso
`'B' < 'a'`	1	verdadero
`'b' > 'Z'`	1	verdadero

Comparar letras es esencial para alfabetizar nombres o usar caracteres para seleccionar una opción particular en situaciones de toma de decisiones. Las cadenas de caracteres también pueden compararse. Por último, pueden compararse dos cadenas de expresiones usando operadores relacionales o los métodos de comparación de la clase `string` (capítulo 7). En el conjunto de caracteres ASCII, un espacio en blanco precede (y es considerado "menor que") a todas las letras y números; las letras del alfabeto son almacenadas en orden de la A a la Z; y los dígitos son almacenados en orden del 0 al 9. En esta secuencia, las letras minúsculas vienen después (son consideradas "mayores que") las letras mayúsculas, y los códigos de las letras vienen después (son "mayores que") los códigos de los dígitos (véase el apéndice B).

Cuando se comparan dos cadenas, sus caracteres individuales se comparan un par a la vez (ambos primeros caracteres, luego ambos segundos caracteres, etc.). Si no se encuentran diferencias, las cadenas son iguales; si se encuentra una diferencia, la cadena con el primer

carácter inferior es considerada la cadena más pequeña. Los siguientes son ejemplos de comparaciones de cadenas:

Expresión	Valor	Interpretación	Comentario
`"Hola"> "Adios"`	`1`	`verdadero`	La primera 'H' en Hola es mayor que la primera 'A' en Adiós
`"SOLANO" > "JIMENES"`	`1`	`verdadero`	La primera 'S' en SOLANO es mayor que la primera 'J' en JIMENEZ
`"123" > "1227"`	`1`	`verdadero`	El tercer carácter, el '3', en 123 es mayor que el tercer carácter, el '2', en 1227.
`"Bejuco" > "Beata"`	`1`	`verdadero`	El tercer carácter, la 'j', en Bejuco es mayor que el tercer carácter 'a' en Beata.
`"Hombre" == "Mujer"`	`0`	`falso`	La primera 'H' en Hombre no es igual a la primera 'M' en Mujer.
`"planta" <"planeta"`	`0`	`falso`	La 't' en planta es mayor que la 'e' en planeta.

Operadores lógicos

Además de usar expresiones relacionales simples como condiciones, pueden crearse condiciones más complejas usando los operadores lógicos AND, OR y NOT. Estos operadores son representados por los símbolos `&&`, `||`, y `!`, respectivamente.

Cuando el operador AND, `&&`, se usa con dos expresiones simples, la condición es verdadera sólo si ambas expresiones individuales son verdaderas por sí mismas. Por tanto, la condición lógica

```
(voltaje > 48) && (miliamperes < 10)
```

es verdadera sólo si `voltaje` es mayor que 48 y `miliamperes` es menor que 10. En vista que los operadores relacionales tienen una precedencia mayor que los operadores lógicos, podrían haberse omitido los paréntesis en esta expresión lógica.

El operador lógico OR, `||`, también se aplica entre dos expresiones. Cuando se usa el operador OR, la condición se satisface si cualquiera de las dos expresiones o ambas son verdaderas. Por tanto, la condición

```
(voltaje > 48) || (miliamperes < 10)
```

es verdadera si `voltaje` es mayor que 48, `miliamperes` es menor que 10, o si ambas condiciones son verdaderas. Una vez más, los paréntesis que rodean a las expresiones relacionales se incluyen para hacer más fácil de leer la expresión. Debido a la mayor precedencia de los operadores relacionales en relación con los operadores lógicos, se haría la misma evaluación aun si se omitieran los paréntesis.

Para las declaraciones

```
int i, j;
double a, b, completo;
```

las siguientes representan condiciones válidas:

```
a > b
(i == j) || (a < b) || completo
(a/b > 5) && (i <= 20)
```

Antes que puedan ser evaluadas estas condiciones, deben conocerse los valores de `a`, `b`, `i`, `j`, y `completo`. Suponiendo que `a` = 12.0, `b` = 2.0, `i` = 15, `j` = 30 y `completo` = 0.0 las expresiones anteriores producirían los siguientes resultados:

Expresión	Valor	Interpretación
`a > b`	`1`	`completo`
`(i == j) || (a < b) || completo`	`0`	`falso`
`(a/b > 5) && (i <= 20)`	`1`	`verdadero`

El operador NOT se usa para cambiar una expresión a su estado opuesto; es decir, si la expresión tiene cualquier valor diferente de cero (verdadero), `!expression` produce un valor de cero (falso). Si una expresión es falsa para comenzar (tiene un valor de cero), `!expression` es verdadero y produce 1. Por ejemplo, asumiendo que el número 26 es almacenado en la variable `edad`, la expresión `edad > 40` tiene un valor de cero (es falsa), mientras la expresión `!(edad > 40)` tiene un valor de 1. En vista que el operador NOT se usa sólo con una expresión, es un operador unitario.

Los operadores relacionales y lógicos tienen una jerarquía de ejecución similar a la de los operadores aritméticos. La tabla 4.2 enumera la precedencia de estos operadores en relación con los otros operadores usados.

Tabla 4.2

Operador	Asociatividad
`!` unitario `-` `++` `--`	derecha a izquierda
`*` `/` `%`	izquierda a derecha
`+` `-`	izquierda a derecha
`<` `<=` `>` `>=`	izquierda a derecha
`==` `!=`	izquierda a derecha
`&&`	izquierda a derecha
`||`	izquierda a derecha
`=` `+=` `-=` `*=` `/=`	derecha a izquierda

El siguiente ejemplo ilustra el uso de la precedencia y asociatividad de un operador para evaluar expresiones relacionales, suponiendo las siguientes declaraciones:

```
char key = 'm';
int i = 5, j = 7, k = 12;
double x = 22.5;
```

Expresión	Expresión equivalente	Valor	Interpretación
`i + 2 == k - 1`	`(i + 2) == (k - 1)`	0	falso
`3 * i - j <  22`	`(3 * i) - j < 22`	1	verdadero
`i + 2 * j > k`	`(i + (2 * j)) > k`	1	verdadero
`k + 3 <= -j + 3 * i`	`(k + 3) <= ((-j) + (3*i))`	0	falso
`'a' + 1 == 'b'`	`('a' + 1) == 'b'`	1	verdadero
`key - 1 >  'p'`	`(key - 1) > 'p'`	0	falso
`key + 1 == 'n'`	`(key + 1) == 'n'`	1	verdadero
`25 >= x + 1.0`	`25 >= (x + 1.0)`	1	verdadero

Como con todas las expresiones, los paréntesis pueden utilizarse para alterar la prioridad del operador asignado y mejorar la legibilidad de las expresiones relacionales. Al evaluar primero las expresiones dentro de los paréntesis, la siguiente condición compuesta se evalúa como

```
(6 * 3 == 36 / 2) || (13 < 3 * 3 + 4) && !(6 - 2 < 5)
       (18 == 18) ||   (13 < 9 + 4)    && !(4 < 5)
                1 ||   (13 < 13)       && !1
                1 ||          0        &&  0
                1 ||          0
                  1
```

Un problema de exactitud numérica

Un problema que puede ocurrir con las expresiones relacionales de C++ es uno de exactitud numérica sutil relacionado con los números de precisión simple y precisión doble. Debido a la forma en que las computadoras almacenan estos números, deberían evitarse las pruebas de igualdad de valores y variables de precisión simple y precisión doble usando el operador relacional ==.

La razón para esto es que muchos números decimales, como 0.1, por ejemplo, no pueden ser representados con exactitud en binario usando un número finito de bits. Por tanto, la prueba de igualdad exacta para estos números puede fallar. Cuando se desea la igualdad de valores no enteros es mejor requerir que el valor absoluto de la diferencia entre operandos sea menor que algún valor extremadamente pequeño. Por tanto, para operandos de precisión simple y precisión doble, la expresión general

operando_1 == operando_2

debería reemplazarse por la condición

abs(operando_1 – operando_2) < 0.000001

donde el valor 0.000001 puede alterarse con cualquier otro valor aceptablemente pequeño. Por tanto, si la diferencia entre los dos operandos es menor que 0.000001 (o cualquier otra

cantidad seleccionada por el usuario), los dos operandos se consideran iguales en esencia. Por ejemplo, si `x` y `y` son variables de precisión simple, una condición como

```
x/y == 0.35
```

debería ser programada como

```
abs(x/y - 0.35) < EPSILON
```

donde `EPSILON` puede ser una constante fijada en cualquier valor aceptablemente pequeño, como $0.000001._2$ Esta última condición asegura que las ligeras inexactitudes al representar números no enteros en binario no afecten la evaluación de la condición probada. En vista que todas las computadoras tienen una representación binaria exacta de cero, las comparaciones para igualdad exacta a cero no encuentran este problema de exactitud numérica.

Ejercicios 4.1

1. Determine el valor de las siguientes expresiones. Suponga `a` = 5, `b` = 2, `c` = 4, `d` = 6, y `e` = 3.
   - **a.** `a > b`
   - **b.** `a != b`
   - **c.** `d % b == c % b`
   - **d.** `a * c != d * b`
   - **e.** `d * b == c * e`
   - **f.** `!(a * b)`
   - **g.** `!(a % b * c)`
   - **h.** `!(c % b * a)`
   - **i.** `b % c * a`
2. Usando paréntesis, vuelva escribir las siguientes expresiones para indicar en forma correcta su orden de evaluación. Luego evalúe cada expresión suponiendo que `a` = 5, `b` = 2 y `c` = 4.
   - **a.** `a % b * c && c % b * a`
   - **b.** `a % b * c || c % b * a`
   - **c.** `b % c * a && a % c * b`
   - **d.** `b % c * a || a % c * b`
3. Escriba expresiones relacionales para denotar las siguientes condiciones (use nombres variables de su elección):
   - **a.** La distancia es igual a 30 pies.
   - **b.** La temperatura ambiente es 86.4.
   - **c.** Una velocidad es 55 MPH.
   - **d.** El mes actual es 12 (diciembre).
   - **e.** La letra introducida es K.
   - **f.** Una longitud es mayor que dos pies y menor que tres pies.
   - **g.** El día actual es el 15o. día del 1er mes.

[2]El uso de la función `abs()` requiere la inclusión del archivo de encabezado `cmath`. Esto se hace colocando la instrucción de preprocesador `#include<cmath>` ya sea inmediatamente antes o después de la instrucción de preprocesador `#include<iostream>`. Los sistemas basados en UNIX también requieren la inclusión específica de la biblioteca matemática en tiempo de compilación con un argumento `-lm` en la línea de comandos.

h. La velocidad del automóvil es 35 MPH y su aceleración es mayor que 4 MPH por segundo.
i. La velocidad de un automóvil es mayor que 50 MPH y se ha estado moviendo al menos por 5 horas.
j. El código es menor que 500 caracteres y toma más de 2 microsegundos en transmitirse.

4. Determine el valor de las siguientes expresiones, suponiendo que `a` = 5, `b` = 2, `c` = 4 y `d` = 5.

a. `a == 5`
b. `b * d == c * c`
c. `d % b * c > 5 || c % b * d < 7`

4.2 La instrucción `if-else`

La estructura `if-else` dirige a la computadora a ejecutar una serie de una o más instrucciones basadas en el resultado de una comparación. Por ejemplo, suponga que se va a calcular el área de un círculo dado el radio como un valor de entrada. Si la entrada es un número negativo se va a imprimir un mensaje indicando que el radio no puede ser un valor negativo; de lo contrario se va a calcular e imprimir el área del círculo. La estructura `if-else` puede utilizarse en esta situación para seleccionar la operación correcta con base en si el radio es negativo o no. La sintaxis general de la instrucción `if-else` es

```
if (expresion)  instruccion1;
else instruccion2;
```

La expresión se evalúa primero. Si el valor de la expresión es diferente de cero, se ejecuta *instruccion1*. Si el valor es cero, se ejecuta la instrucción después de la palabra clave else. Por tanto, siempre se ejecuta una de las dos instrucciones (ya sea *instruccion1* o *instruccion2*, pero no ambas) dependiendo del valor de la expresión. Hay que observar que la expresión probada debe ponerse entre paréntesis y que se coloca un punto y coma después de cada instrucción

Por claridad, la instrucción `if-else` se escribe de manera típica en cuatro líneas usando la forma

```
if (expresion) <-------- no va punto y coma aquí
    instruccion1;
else <------------------ no va punto y coma aquí
   instruccion2;
```

La forma de la instrucción `if-else` que se selecciona de manera típica depende del largo de las instrucciones 1 y 2. Sin embargo, cuando se usa la segunda forma, no se pone un punto y coma después del paréntesis o de la palabra clave `else`. Los puntos y comas sólo van después del final de las instrucciones.

El diagrama de flujo para la instrucción `if-else` se muestra en la figura 4.2.

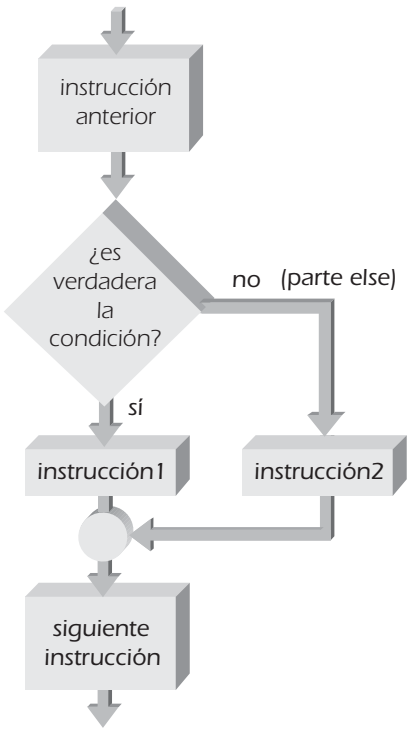


Figure 4.2 *El diagrama de flujo de* `if-else`.

Como un ejemplo específico de una estructura `if-else`, se eleborará un programa C++ para determinar el área de un círculo examinando primero el valor del radio. La condición que se va a probar es si el radio es menor que cero. Una instrucción `if-else` apropiada para esta situación es:

```
if (radio < 0.0)
  cout << "Un radio negativo es invalido" << endl;
else
  cout << "El area de este circulo es " << 3.1416 * pow(radius,2) << endl;
```

Aquí se ha usado el operador relacional < para representar la relación "menor que". Si el valor del radio es menor que 0, la condición es verdadera (tiene un valor de 1) y la instrucción `cout >> "Un radio negativo es invalido";` se ejecuta. Si la condición no es verdadera, el valor de la expresión es cero, y se ejecuta la instrucción posterior a la palabra clave `else`. El programa 4.1 ilustra el uso de esta instrucción en un programa completo.

Programa 4.1

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  double radio;

  cout << "Por favor introduzca el radio: ";
  cin  >> radio;

  if (radio < 0.0 )
    cout << "Un radio negativo es invalido" << endl;
  else
    cout << "El area de este circulo es " << 3.1416 * pow(radio,2) << endl;

  return 0;
}
```

Se insertó una línea en blanco antes y después de la instrucción `if-else` para resaltarla en el programa completo. Se continuará haciendo esto a lo largo del texto para enfatizar la instrucción que se está presentando.

Para ilustrar la selección en acción, el programa 4.1 se ejecutó dos veces con diferentes datos de entrada. Los resultados fueron

```
Por favor introduzca el radio: -2.5
Un radio negativo es invalido
```

y:

```
Por favor introduzca el radio: 2.5
El area de este circulo es 19.635
```

Al revisar esta salida, observe que el radio en la primera ejecución del programa fue menor que 0 y la parte `if` de la estructura `if-else` ejecutó de manera correcta la instrucción `cout`, indicando al usuario que un radio negativo es inválido. En la segunda ejecución, el radio no es negativo y la parte `else` de la estructura `if-else` se usó para producir un cálculo correcto del área de

$$3.1416 * (2.5)^2 = 19.635$$

Aunque cualquier expresión puede ser probada con una instrucción `if-else`, se usan de manera predominante expresiones relacionales. Sin embargo, instrucciones como

```
if (num)
  cout << "¡Loteria!";
```

```
else
  cout << "¡Perdiste!";
```

son válidas. En vista que `num`, en sí misma, es una expresión válida, el mensaje `¡Loteria!` es desplegado si `num` tiene cualquier valor diferente de cero y el mensaje `¡Perdiste!` se despliega si `num` tiene un valor de cero.

Instrucciones compuestas

Aunque nada más se permite una sola instrucción en las partes `if` y `else` de la instrucción `if-else`, esta instrucción puede ser una sola instrucción compuesta. Una ***instrucción compuesta*** es una secuencia de instrucciones individuales contenidas entre llaves, como se muestra en la figura 4.3.

```
{
   instruccion1;
   instruccion2;
   instruccion3;
        .
        .
        .
   ultima instruccion;
}
```

Figura 4.3 *Una instrucción compuesta consiste de instrucciones individuales encerradas entre llaves.*

El uso de llaves para encerrar un conjunto de instrucciones individuales crea un solo bloque de instrucciones, el cual puede usarse en cualquier parte en un programa en C++ en lugar de una sola instrucción. El siguiente ejemplo ilustra el uso de una instrucción compuesta dentro de la forma general de una instrucción `if-else`.

```
if (expresion)
{
  instruccion1; // pueden colocarse dentro de las llaves
  instruccion2; // tantas instrucciones como sean
                   necesarias
  instruccion3; // cada instruccion debe terminar con un;
}
else
{
  instruccion4;
  instruccion5;
       .
       .
  instruccion_n;
}
```

El programa 4.2 ilustra el uso de una instrucción compuesta en un programa real.

Programa 4.2

```
#include <iostream>
#include <iomanip>
using namespace std;

// un programa para conversion de temperaturas
int main()
{
  char tipo_temp;
  double temp, fahren, celsius;

  cout << "Introduzca la temperatura que se va a convertir: ";
  cin  >> temp;
  cout << "Introduzca una f si la temperatura es en grados Fahrenheit";
  cout << "\n o una c si la temperatura es en grados Celsius: ";
  cin  >> tipo_temp;

    // establecer los formatos de salida
  cout << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);

  if (tipo_temp == 'f')
  {
    celsius = (5.0 / 9.0) * (temp - 32.0);
    cout << "\nLa temperatura Celsius equivalente es "
         << celsius << endl;
  }
  else
  {
    fahren =  (9.0 / 5.0) * temp + 32.0;
    cout << "\nLa temperatura Fahrenheit equivalente es "
         << fahren << endl;
  }

  return 0;
}
```

El programa 4.2 verifica si el valor en `tipo_temp` es f. Si el valor es f, se ejecuta la instrucción compuesta correspondiente a la parte `if` de la instrucción `if-else`. Cualquier otra letra produce la ejecución de la instrucción compuesta correspondiente a la parte `else`. A continuación se presenta una muestra de ejecución del programa 4.2.

```
Introduzca la temperatura que se va a convertir: 212
Introduzca una f si la temperatura es en grados Fahrenheit
o una c si la temperatura es en grados Celsius: f

La temperatura Celsius equivalente es 100.00
```

El alcance de un bloque

Todas las instrucciones contenidas dentro de una instrucción compuesta constituyen un solo bloque de código y cualquier variable declarada dentro de dicho bloque sólo tiene significado entre su declaración y las llaves de cierre que definen el bloque. Por ejemplo, considere la siguiente sección de código, la cual consta de dos bloques de código:

```
{   // comienzo del bloque exterior
   int a = 25;
   int b = 17;

   cout << "El valor de a es " << a
        <<" and b is " << b << endl;

   {    // comienzo del bloque interior
     float a = 46.25;
     int c = 10;

     cout << "a es ahora " << a
          << " b es ahora " << b
          << " y c es " << c << endl;
   }   // fin del bloque interior

   cout << "a es ahora " << a
        << " y b es " << b << endl;

}   // fin del bloque exterior
```

La salida que produce esta sección de código es:

```
El valor de a es 25 y b es 17
a es ahora 46.25 b es ahora 17 y c es 10
a es ahora 25 y b es 17
```

Esta salida se produce como sigue: El primer bloque de código define dos variables nombradas `a` y `b`, las cuales pueden usarse en cualquier parte dentro de este bloque después de su declaración, incluyendo cualquier bloque contenido dentro de éste. Dentro del bloque interior, se han declarado dos variables nuevas, nombradas `a` y `c`. En esta etapa entonces, se han declarado cuatro variables diferentes, dos de las cuales tienen el mismo nombre. Cualquier variable referenciada primero produce un intento de acceso a una variable declarada en forma correcta dentro del bloque que contiene la referencia. Si no se define una variable dentro del bloque, se hace un intento de acceso a una variable en el siguiente bloque inmediato exterior hasta que resulta un acceso válido.

Punto de Información

Colocación de llaves en una instrucción compuesta

Una práctica común para algunos programadores en C++ es colocar la llave de apertura de una instrucción compuesta en la misma línea que las instrucciones `if` y `else`. Usando esta convención, la instrucción `if` en el programa 4.2 aparecería como se muestra a continuación. (Esta colocación sólo es una cuestión de estilo, ambos estilos se usan y ambos son aceptables.)

```
if (tipo_temp == 'f') {
  celsius = (5.0 / 9.0) * (temp - 32.0);
  cout << "\nLa temperatura Celsius equivalente es "
       << celsius << endl;
}
else {
  fahren = (9.0 / 5.0) * temp + 32.0;
  cout << "\nLa temperatura Fahrenheit equivalente es "
       << celsius << endl;
}
```

Por tanto, los valores de las variables `a` y `c` referenciadas dentro del bloque interior usan los valores de las variables `a` y `c` declaradas en ese bloque. En vista que ninguna variable nombrada `b` fue declarada dentro del bloque interior, el valor de `b` desplegado desde dentro del bloque interior se obtiene del bloque exterior. Por último, el último objeto `cout`, el cual está fuera del bloque interior, despliega el valor de la variable `b` declarada en el bloque exterior. Si se hiciera un intento por desplegar el valor de `c` en cualquier parte en el bloque exterior, el compilador emitiría un mensaje de error declarando que `c` es un símbolo no definido.

El área dentro de un programa donde una variable puede utilizarse se conoce de manera formal como el **alcance** de la variable, y se ampliará este tema en el capítulo 6.

Selección unidireccional

Una modificación útil de la instrucción `if-else` implica omitir por completo la parte `else` de la instrucción. En este caso, la instrucción `if` adquiere la forma abreviada y con frecuencia útil

```
if (expresion)
  instruccion;
```

La instrucción que sigue a `if (expresion)` sólo se ejecuta si la expresión tiene un valor diferente de cero (una condición verdadera). Como antes, la instrucción puede ser una instrucción compuesta. El diagrama de flujo para esta instrucción se ilustra en la figura 4.4.

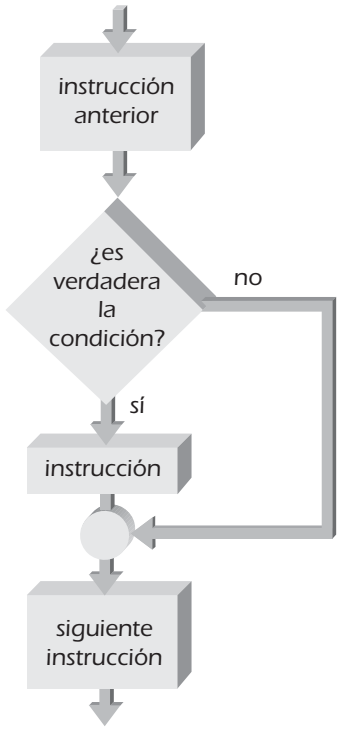


Figura 4.4 *Diagrama de flujo para la instrucción* `if` *unidireccional.*

Esta forma modificada de la instrucción `if` se llama instrucción `if` unidireccional. El programa 4.3 usa esta instrucción para desplegar de manera selectiva un mensaje para automóviles que han sido conducidos más de 3000.0 millas.

Programa 4.3

```
#include <iostream>
using namespace std;

int main()
{
  const double LIMITE = 3000.0;
  int num_id;
  double millas;

  cout << "Por favor introduzca numero de automovil y millas recorridas: ";
  cin  >> num_id >> millas;

  if(millas > LIMITE)
    cout << "  El automovil " << num_id << " esta arriba del limite.\n";

  cout << "Fin de la salida del programa.\n";

  return 0;
}
```

Como una ilustración de estos criterios de selección unidireccionales en acción, el programa 4.3 se ejecutó dos veces, cada vez con diferentes datos de entrada. Sólo los datos de entrada para la primera ejecución causan que se despliegue el mensaje `El automovil 256 esta arriba del limite`.

```
Por favor introduzca numero de automovil y millas
  recorridas: 256 3562.8
   El automovil 256 esta arriba del limite.
Fin de la salida del programa.
```

y

```
Por favor introduzca numero de automovil y millas
  recorridas: 23 2562.3
Fin de la salida del programa.
```

Problemas asociados con la instrucción `if-else`

Dos de los problemas más comunes que se pueden enfrentar al empezar a usar la instrucción `if-else` de C++ son:

1. Comprender mal las implicaciones completas de lo que es una expresión y
2. Usar el operador de asignación, =, en lugar del operador relacional ==.

Recuerde que una expresión es cualquier combinación de operandos y operadores que produce un resultado. Esta definición es demasiado amplia y abarca más de lo que parece al principio. Por ejemplo, todas las siguientes son expresiones C++ válidas:

```
edad + 5
edad = 30
edad == 40
```

Suponiendo que las variables son declaradas de manera adecuada, cada una de las expresiones anteriores produce un resultado. El programa 4.4 usa el objeto `cout` para desplegar el valor de estas expresiones cuando `edad` = 18.

Programa 4.4

```
#include <iostream>
using namespace std;

int main()
{
  int edad = 18;

  cout << "El valor de la primera expresion es " << (edad + 5) << endl;
  cout << "El valor de la segunda expresion es " << (edad = 30) << endl;
  cout << "El valor de la tercera expresion es " << (edad == 40) << endl;

  return 0;
}
```

El despliegue producido por el programa 4.4 es:

```
El valor de la primera expresion es 23
El valor de la segunda expresion es 30
El valor de la tercera expresion es 0
```

Como ilustra la salida del programa 4.4, cada expresión, por sí misma, tiene un valor asociado con ella. El valor de la primera expresión es la suma de la variable `edad` más 5, el cual es 23. El valor de la segunda expresión es 30, el cual también es asignado a la variable `edad`. El valor de la tercera expresión es cero, en vista que `edad` no es igual a 40, y una condición falsa se representa en C++ con un valor de cero. Si el valor en edad hubiera sido 40, la expresión relacional a == 40 hubiera sido verdadera y habría tenido un valor de 1.

Punto de Información

El tipo de datos booleano

Antes del estándar actual ANSI/ISO de C++, éste no tenía un tipo de datos booleano incorporado con sus dos valores booleanos, verdadero y falso. En vista que este tipo de datos originalmente no era parte del lenguaje, una expresión probada podía no evaluarse como un valor booleano. Por tanto, la sintaxis

if(la expresión booleana es verdadera)
ejecute esta instrucción;

tampoco estaba incorporada en C o en C++. Más bien, tanto C como C++ usan la sintaxis que abarca más:

if(expresión)
ejecute esta instrucción;

donde *expresión* es cualquier expresión que produzca un valor numérico. Si el valor de la expresión probada es un valor diferente de cero se considera como verdadero, y sólo un valor de cero es considerado falso.

Como lo especifica el estándar ANSI/ISO de C++, éste tiene un tipo de datos booleano incorporado que contiene los dos valores, **verdadero** y **falso**. Las variables booleanas serán declaradas usando la palabra clave **bool**. Como se ponen en práctica en la actualidad, los valores reales representados por los dos valores booleanos, **verdadero** y **falso**, son los valores enteros 1 y 0, respectivamente. Por ejemplo, considere el siguiente programa, el cual declara dos variables booleanas.

```
#include <iostream>
using namespace std;
int main()
{
  bool t1, t2;
  t1 = verdadero;
  t2 = falso;
  cout << "El valor de t1 es " << t1
       << "\ny el valor de t2 es " << endl;
  return 0;
}
```

La salida producida por este programa es:

```
El valor de t1 es 1
y el valor de t2 es 0
```

Como se puede observar en la salida, los valores booleanos **verdadero** y **falso** están representados por los valores enteros 1 y 0, respectivamente. Los valores booleanos verdadero y falso tienen las siguientes relaciones.

```
!true= es falso
!false= es verdadero
```

Además, aplicar un operador ++ como sufijo o prefijo a una variable del tipo bool establecerá el valor booleano en **verdadero**. Los operadores – – de sufijo y prefijo no pueden aplicarse a variables booleanas.

Los valores booleanos también pueden compararse, como se ilustra en el siguiente código:

```
if (t1 == t2)
  cout << "Los valores son iguales" << endl;
else
  cout << "Los valores no son iguales" << endl;
```

Por último, asignar cualquier valor diferente de cero a una variable booleana produce que la variable se establezca como **verdadera**; es decir, un valor de 1; y asignar un valor de cero a una variable booleana produce que la variable se establezca como **falsa**; es decir, un valor de 0.

Ahora suponga que se pretende usar la expresión relacional `edad == 40` en la instrucción `if`

```
if (edad == 40)
  cout << "¡Feliz Cumpleaños!";
```

pero se escribió mal como `edad = 40`, lo que resulta en

```
if (edad = 40)
  cout << "¡Feliz Cumpleaños!";
```

En vista que el error da como resultado una expresión C++ válida, y cualquier expresión de C++ puede ser probada por una instrucción `if`, la instrucción `if` resultante es válida y causará que se imprima el mensaje `¡Feliz Cumpleaños!` sin importar cuál valor se haya asignado antes a edad. ¿Puede ver por qué?

La condición probada por la instrucción `if` no compara el valor en edad con el número 40, sino que asigna el número `40` a `edad`. Es decir, la expresión `edad = 40` no es una expresión relacional en absoluto, sino una expresión de asignación. Al completar la asignación la expresión en sí tiene un valor de 40. En vista que C++ trata cualquier valor diferente de cero como verdadero, se ejecuta la instrucción `cout`. Otra forma de ver esto es darse cuenta que la instrucción `if` es equivalente a las dos instrucciones siguientes:

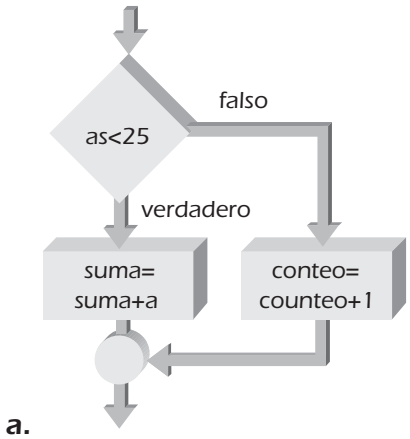
```
edad = 40;      // asigna 40 a edad
if (edad)       // prueba el valor de edad
  cout << "¡Feliz Cumpleaños!";
```

En vista que un compilador de C++ no tiene medio de saber que la expresión que se está probando no es la deseada, debe tenerse especial cuidado cuando se escriben condiciones.

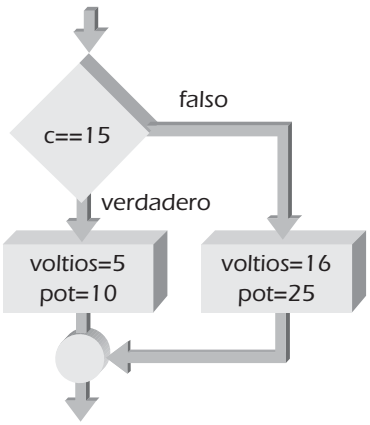
Ejercicios 4.2

1. Escriba instrucciones `if` apropiadas para cada una de las siguientes condiciones:
   - **a.** Si un ángulo es igual a 90 grados imprima el mensaje "El ángulo es un ángulo recto", de lo contrario imprima el mensaje "El ángulo no es un ángulo recto".
   - **b.** Si la temperatura está por encima de 100 grados desplegar el mensaje "arriba del punto de ebullición del agua", de lo contrario desplegar el mensaje "abajo del punto de ebullición del agua".
   - **c.** Si el número es positivo sumar el número a `sumpos`, si no sume el número a `sumneg`.
   - **d.** Si la pendiente es menor que 0.5 fijar la variable `flag` en cero, de lo contrario fijar `flag` en uno.
   - **e.** Si la diferencia entre `voltios1` y `voltios2` es menor que 0.001, fijar la variable `aprox` en cero, de lo contrario calcular `aprox` como la cantidad (`voltios1 – voltios2`) / 2.0.
   - **f.** Si la frecuencia es superior a 60, desplegar el mensaje "La frecuencia es demasiado alta".
   - **g.** Si la diferencia entre `temp1` y `temp2` excede 2.3, calcular `error` como `(`temp1 —temp2`) * factor.`
   - **h.** Si *x* es mayor que *y* y *z* es menor que 20, leer un valor para *p*.
   - **i.** Si la distancia es mayor que 20 y es menor que 35, leer un valor para tiempo.

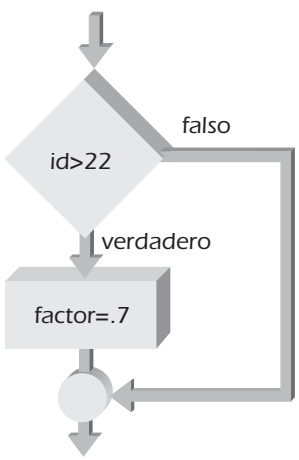
2. Escriba instrucciones `if` correspondientes a las condiciones ilustradas por cada uno de los siguientes diagramas de flujo.



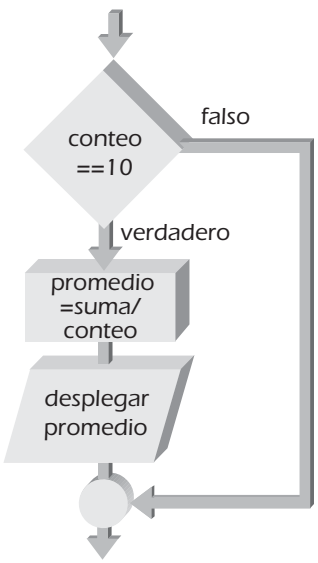
a.



b.



c.



d.

3. Escriba un programa en C++ que le pida al usuario que introduzca dos números. Si el primer número introducido es mayor que el segundo número el programa deberá imprimir el mensaje "El primer número es mayor", de lo contrario deberá imprimir el mensaje "El primer número es menor". Pruebe su programa introduciendo los números 5 y 8 y luego usando los números 11 y 2. ¿Qué piensa que desplegará su programa si los dos números introducidos son iguales? Pruebe este caso.

4. ***a.*** Una cierta onda es de 0 voltios para un tiempo menor que 2 segundos y de 3 voltios para un tiempo igual o mayor que 2 segundos (estas ondas se conocen como funciones de paso). Escriba un programa en C++ que acepte tiempo en la variable nombrada `tiempo` y despliegue el voltaje apropiado dependiendo del valor de entrada.

b. ¿Cuantas ejecuciones debería hacer para el programa escrito en el ejercicio 4a para verificar que opera en forma correcta? ¿Qué datos debería introducir en cada una de las ejecuciones del programa?

5. Una prueba de aislamiento para un cable requiere que el aislamiento resista al menos 600 voltios. Escriba un programa en C++ que acepte una prueba de voltaje e imprima el mensaje "PRUEBA DE VOLTAJE APROBADA" o el mensaje "PRUEBA DE VOLTAJE NO APROBADA", según sea apropiado.

6. a. Escriba un programa en C++ para calcular el valor de la presión en libras por pulgada cuadrada (psi) de una onda descrita como sigue:

Para tiempo, t, igual a o menor que 35 segundos, la presión es $0.46t$ psi y para tiempo mayor que 35 segundos la presión es $0.19t + 9.45$ psi.

El programa deberá solicitar el tiempo como entrada y deberá desplegar la presión como salida.

b. ¿Cuántas ejecuciones deberá hacer para el programa escrito en el ejercicio 6a para verificar que funciona en forma correcta? ¿Qué datos debería introducir en cada una de las ejecuciones del programa?

7. a. Escriba un programa en C++ que despliegue el mensaje "PROCEDER CON EL DESPEGUE" o "CANCELAR EL DESPEGUE" dependiendo de la entrada. Si el carácter g es introducido en la variable codigo, deberá desplegarse el primer mensaje; de lo contrario deberá desplegarse el segundo mensaje.

b. ¿Cuántas ejecuciones deberá hacer para el programa escrito en el ejercicio 7a para verificar que opera en forma correcta? ¿Qué datos debería introducir en cada una de las ejecuciones del programa?

8. Una fábrica pequeña genera su propia energía con un generador de 20 kilowatts y un generador de 50 kilowatts. El gerente de la planta indica cuál generador se requiere al introducir un código de carácter. Escriba un programa en C++ que acepte este código como entrada. Si se introduce el código s deberá desplegarse un mensaje que le indique al capataz de la planta que use el generador más pequeño; de lo contrario deberá ser la salida un mensaje que le indique el uso del generador más grande.

4.3 INSTRUCCIONES `if` ANIDADAS

Como se ha visto, una instrucción `if-else` puede contener instrucciones simples o compuestas. Puede usarse cualquier instrucción de C++ válida, incluyendo otra instrucción `if-else`. Por tanto, pueden incluirse una o más instrucciones `if-else` dentro de cualquier parte de una instrucción `if-else`. La inclusión de una o más instrucciones `if` dentro de una instrucción `if` se llama instrucción `if` *anidada*. Por ejemplo, sustituyendo la instrucción `if` unidireccional

```
if (distancia > 500)
  cout << "oprima";
```

por la instruccion1 en la siguiente instrucción if

```
if (horas < 9)
  instruccion1;
else
  cout << "suelte";
```

resulta la instrucción if anidada

```
if (horas < 9)
{
  if (distancia > 500)
    cout << "oprima";
}
else
  cout << "suelte";
```

Las llaves que rodean al if unidireccional interior son esenciales debido a que en su ausencia C++ asocia un else con el if más cercano que no tenga complemento. Por tanto, sin las llaves, la instrucción anterior es equivalente a

```
if (horas < 9)
  if (distancia > 500)
    cout << "oprima";
  else
    cout << "suelte";
```

Aquí else se empareja con el if interior, lo cual destruye el significado de la instrucción if-else original. Hay que observar también que la sangría es irrelevante en lo que respecta al compilador. Exista la sangría o no, *la instrucción es compilada al asociar el último* else *con el* if *más cercano no emparejado, a menos que se usen llaves para alterar el emparejamiento por omisión.*

El proceso de anidar instrucciones if puede extenderse de manera indefinida, de modo que la instrucción cout << "oprima"; podría reemplazarse por una instrucción if-else completa o por otra instrucción if unidireccional.

La figura 4.5 ilustra la forma general de una instrucción if-else anidada cuando una instrucción if-else es anidada a) dentro de la parte if de una instrucción if-else y b) dentro de la parte else de una instrucción if-else.

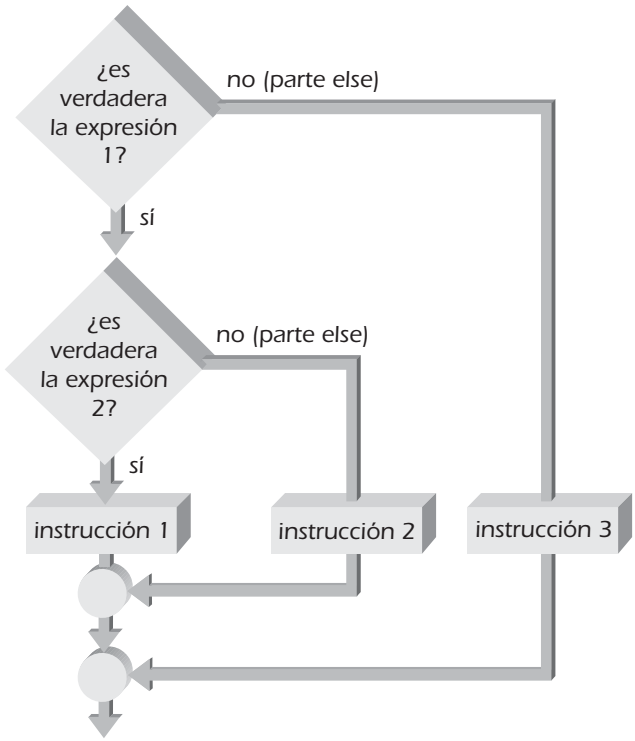


Figura 4.5a *Instrucción* `if-else` *anidada en una* `if`.

La cadena `if-else`

En general, la anidación ilustrada en la figura 4.5a tiende a ser confusa y es mejor evitarla en la práctica. Sin embargo, ocurre una construcción muy útil para la anidación ilustrada en la figura 4.5b, la cual tiene la forma

```
if (expresion_1)
  instruccion1;
else
  if (expresion_2)
    instruccion2;
  else
    instruccion3;
```

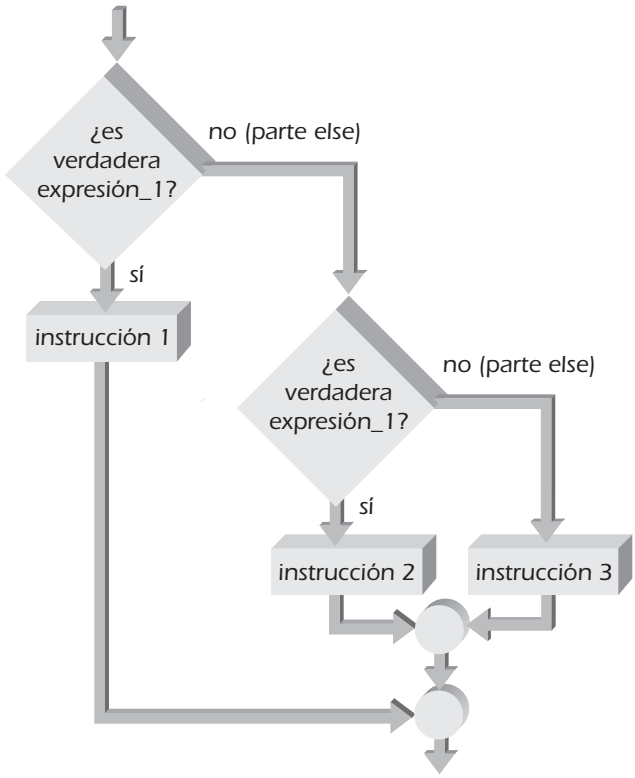


Figura 4.5b *Instrucción* `if-else` *anidada en un* `else`.

Como con todos los programas en C++, en vista que se ignora el espacio en blanco, la sangría mostrada no se requiere. De manera más típica, la construcción anterior se escribe usando el siguiente arreglo:

```
if (expresion_1)
  instruccion1;
else if (expression_2)
  instruccion2;
else
  instruccion3;
```

Esta forma de una instrucción `if` anidada es muy útil en la práctica, y de manera formal se conoce como `cadena if-else`. Cada condición es evaluada en orden, y si cualquier condición es verdadera se ejecuta la instrucción correspondiente y el resto de la cadena se termina. La instrucción asociada con el `else` final sólo se ejecuta si ninguna de las condiciones anteriores se satisface. Esto sirve como un caso por omisión o aplicado a todos los casos que es útil para detectar una condición imposible o errónea.

La cadena puede continuarse en forma indefinida al hacer de manera repetida que la última instrucción sea otra instrucción `if-else`. Por tanto, la forma general de una cadena `if-else` es:

```
if (expresion_1)
  instruccion1;
else if (expresion_2)
  instruccion2;
else if (expresion_3)
  instruccion3;
      .
      .
      .
else if (expresion_n)
  instruccion_n;
else
  ultima_instruccion;
```

Cada condición es evaluada en el orden en que aparece en la instrucción. Para la primera condición que es verdadera, se ejecuta la instrucción correspondiente, y el resto de las instrucciones en la cadena no se ejecutan. Por tanto, si `expresion_1` es verdadera, sólo se ejecuta `instruccion1`; de lo contrario se prueba `expresion_2`. Si `expresion_2` es verdadera entonces, sólo se ejecuta `instruccion2`; de lo contrario se prueba `expresion_3`, y así en forma sucesiva. El `else` final en la cadena es opcional, y `ultima_instruccion` sólo se ejecuta si ninguna de las expresiones previas fue verdadera.

Para ilustrar el uso de la cadena `if-else`, el programa 4.5 despliega el estado de especificación de cada elemento correspondiente a una letra introducida. Se usan los siguientes códigos de letra:

Estado de la especificación	Código de la entrada
Exploración espacial	`S`
Grado militar	`M`
Grado comercial	`C`
Grado juguete	`T`

Programa 4.5

```
#include <iostream>
using namespace std;

int main()
{
  char codigo;

  cout << "Introduzca un codigo de especificacion: ";
  cin  >> codigo;

  if (codigo == 'S')
      cout << "El elemento tiene grado de exploracion espacial.";
  else if (codigo == 'M')
    cout << "El elemento tiene grado militar.";
  else if (code == 'C')
    cout << "El elemento tiene grado comercial.";
  else if (codigo == 'T')
      cout << "El elemento tiene grado de juguete.";
  else
      cout << "Se ha introducido un codigo invalido.";
  cout << endl;

  return 0;
}
```

Como un ejemplo más de una cadena `if-else`, se determina la salida de una unidad convertidora digital usando la siguiente relación entrada/salida:

Peso de entrada	Lectura de salida
mayor que o igual a 90 lbs	1111
menor que 90 lbs pero mayor que 80 lbs	1110
menor que 80 lbs pero mayor que o igual a 70 lbs	1101
menor que 70 lbs pero mayor que o igual a 60 lbs	1100
menor que 60 lbs	1011

Las siguientes instrucciones pueden usarse para determinar la lectura de salida correcta, donde la variable `inlbs` se usa para almacenar la lectura de entrada:

```
if (inlbs >= 90)
    digout = 1111;
```

```
else if (inlbs >= 80)
    digout = 1110;
else if (inlbs >= 70)
    digout = 1101;
else if (inlbs >= 60)
    digout = 1100;
else
    digout = 1011;
```

Hay que observar que este ejemplo aprovecha el hecho que la cadena se detiene una vez que se encuentra una condición verdadera. Esto se logra verificando primero la entrada de peso mayor. Si el valor de entrada es menor que 90, la cadena `if-else` continúa verificando el siguiente peso más alto, y así en forma sucesiva, hasta que se obtiene la categoría de peso correcta.

El programa 4.6 usa una cadena `if-else` para calcular y desplegar la lectura de salida correcta correspondiente a la entrada de peso en la instrucción `cin`.

Programa 4.6

```
#include <iostream>
using namespace std;

int main()
{
  int digout;
  double inlbs;

  cout << "Introduzca el peso: ";
  cin  >> inlbs;

  if (inlbs >= 90)
      digout = 1111;
  else if (inlbs >= 80)
      digout = 1110;
  else if (inlbs >= 70)
    digout = 1101;
  else if (inlbs >= 60)
    digout = 1100;
  else
    digout = 1011;

  cout << "La salida digital es " << digout << endl;

  return 0;
}
```

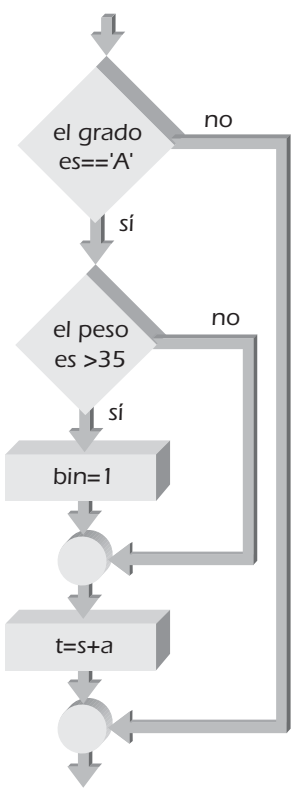
Una muestra de ejecución usando el programa 4.6 se ilustra a continuación.

```
Introduzca el peso: 72.5
La salida digital es 1101
```

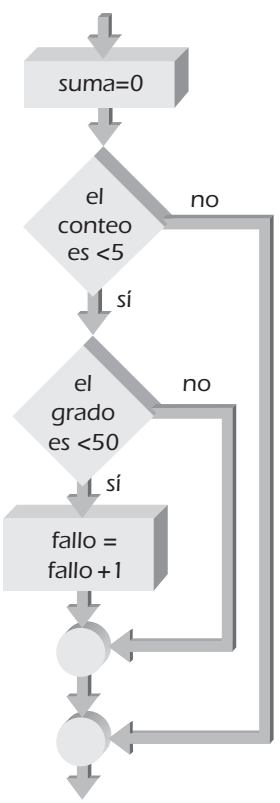
Como con todas las instrucciones de C++, cada instrucción individual dentro de una cadena `if-else` puede reemplazarse por una instrucción compuesta encerrada entre las llaves `{` y `}`.

Ejercicios 4.3

1. Modifique el programa 4.5 para que acepte letras minúsculas y mayúsculas como códigos. Por ejemplo, si un usuario introduce una m o una M, el programa deberá desplegar el mensaje "El elemento tiene grado militar."
2. Escriba instrucciones `if` anidadas correspondientes a las condiciones ilustradas en cada uno de los siguientes diagramas de flujo.



a.



b.

3. Un ángulo es considerado agudo si es menor que 90 grados, obtuso si es mayor que 90 grados y ángulo recto si es igual a 90 grados. Usando esta información, escriba un programa en C++ que acepte un ángulo, en grados, y despliegue el tipo de ángulo correspondiente a los grados introducidos.

4. El nivel de grado de los estudiantes universitarios se determina de manera típica de acuerdo con la siguiente tabla:

Número de créditos completados	Grado
menor que 32	primer año
32 a 63	segundo año
64 a 95	tercer año
96 o más	último año

Usando esta información, escriba un programa en C++ que acepte el número de créditos que ha completado un estudiante, determine el grado del estudiante y lo despliegue.

5. La letra que representa las calificaciones de un estudiante se calcula de acuerdo con la siguiente tabla:

Calificación numérica	Letra
mayor que o igual a 90	A
menor que 90 pero mayor que o igual a 80	B
menor que 80 pero mayor que o igual a 70	C
menor que 70 pero mayor que o igual a 60	D
menor que 60	F

Usando esta información, escriba un programa en C++ que acepte la calificación numérica de un estudiante, convierta la calificación numérica a su calificación en letra equivalente y despliegue la letra.

6. La tolerancia de componentes críticos en un sistema se determina por la aplicación de acuerdo con la siguiente tabla:

Estado de la especificación	Tolerancia
Exploración espacial	Menor que 0.1%
Grado militar	Mayor que o igual a 0.1% y menor que 1%
Grado comercial	Mayor que o igual a 1% y menor que 10%
Grado de juguete	Mayor que o igual a 10%

Usando esta información, escriba un programa en C++ que acepte la lectura de tolerancia de un componente y determine la especificación que debería asignarse al componente.

7. Escriba un programa en C++ que acepte un número seguido por un espacio y luego una letra. Si la letra que sigue al número es f, el programa tratará al número introducido como una temperatura en grados Fahrenheit, convertirá el número a los grados Celsius equivalentes y desplegará un mensaje adecuado. Si la letra que sigue al número es c, el programa tratará al número introducido como una temperatura en Celsius, convertirá el número a los grados Fahrenheit equivalentes y desplegará un mensaje adecuado. Si la letra no es f ni c, el programa imprimirá el mensaje que los datos introducidos son incorrectos y terminará. Use una cadena `if-else` en su programa y use las fórmulas de conversión:

Celsius = (5.0 / 9.0) * (*Fahrenheit* – 32.0)
Fahrenheit = (9.0 / 5.0) * *Celsius* + 32.0

8. Usando las relaciones del programa 4.6, el siguiente programa calcula la salida digital:

```
int main()
{
  int digout;
  double inlbs;

  cout << "Introduzca el peso: ";
  cin  >> inlbs;

  if (inlbs >= 90) digout = 1111;
  if (inlbs >= 80) && (inlbs <= 90) digout = 1110;
  if (inlbs >= 70) && (inlbs <= 80) digout = 1101;
  if (inlbs >= 60) && (inlbs <= 70) digout = 1100;
  if (inlbs < 1000) digout = 1011;

  cout << "La salida digital es " << digout << endl;

  return 0;
}
```

a. ¿Este programa producirá la misma salida que el programa 4.6?
b. ¿Cuál programa es mejor y por qué?

9. El siguiente programa fue escrito para producir el mismo resultado que el programa 4.6:

```
int main()
{
  int digout;
  double inlbs;

  cout << "Introduzca el peso: ";
  cin  >> inlbs;
```

```
    if (inlbs < 60)
        digout = 1011;
    else if (inlbs >= 60)
        digout = 1100;
    else if (inlbs >= 70)
      digout = 1101;
    else if (inlbs >= 80)
      digout = 1110;
    else if (inlbs >= 90)
      digout = 1111;

    cout << "La salida digital es " << digout << endl;

    return 0;
}
```

a. ¿Se ejecutará este programa?
b. ¿Qué hace este programa?
c. ¿Para cuáles valores de libras introducidos calculará este programa la salida digital correcta?

4.4 LA INSTRUCCIÓN `switch`

La cadena `if-else` se usa en aplicaciones de programación donde un conjunto de instrucciones debe ser seleccionada entre muchas alternativas posibles. La instrucción `switch` proporciona una alternativa a la cadena `if-else` para casos que comparan el valor de una expresión de número entero con un valor específico. La forma general de una instrucción `switch` es

```
switch (expresion)
{    // inicio de instrucción compuesta
  case valor_1:  <-- termina con dos puntos
    instruccion1;
    instruccion2;
        .
        .
    break;
  case valor_2:  <-- termina con dos puntos
    instruccion;
    instruccion;
        .
        .
    break;
   .
   .
```

```
  case valor_n: ◄──────────── termina con dos puntos
    instruccionw;
    instruccionx;
        .
        .
    break;
  default:      ◄──────────── termina con dos puntos
    instruccionaa;
    instruccionbb;
        .
}    // fin de switch y de la instrucción compuesta
```

La instrucción `switch` usa cuatro palabras clave nuevas: `switch`, `case`, `break` y `default`. Veamos lo que hace cada una de estas palabras.

La palabra clave `switch` identifica el inicio de la instrucción `switch`. La expresión entre paréntesis que sigue a la palabra es evaluada y el resultado de la expresión comparado con diversos valores alternativos contenidos dentro de la instrucción compuesta. La expresión en la instrucción `switch` deben evaluar un resultado en número entero o resultará un error de compilación.

Dentro de la instrucción `switch`, la palabra clave `case` se usa para identificar o etiquetar valores individuales que se comparan con el valor de la expresión `switch`. El valor de la expresión `switch` se compara con cada uno de estos valores `case` en el orden en que se enlistan estos valores hasta que se encuentra una correspondencia. Cuando los valores corresponden, la ejecución comienza con la instrucción que sigue inmediatamente a la correspondencia. Por tanto, como se ilustra en la figura 4.6, el valor de la expresión determina en qué parte de la instrucción `switch` comienza en realidad la ejecución.

Una instrucción `switch` puede contener cualquier número de etiquetas `case`, en cualquier orden. Sin embargo, si el valor de la expresión no corresponde con ninguno de los valores `case`, no se ejecuta ninguna instrucción a menos que se encuentre la palabra clave `default`. La palabra `default` es opcional y opera igual que la última `else` en una cadena `if-else`. Si el valor de la expresión no corresponde con ninguno de los valores `case`, la ejecución del programa comienza con la instrucción que sigue a la palabra `default`.

Una vez que la instrucción `switch` ha localizado un punto de entrada, todas las demás evaluaciones `case` son ignoradas y la ejecución continúa hasta el final de la instrucción compuesta a menos que se encuentre una instrucción `break`. Ésta es la razón por la que la instrucción `break`, la cual identifica el fin de un `case` particular y causa una salida inmediata de la instrucción `switch`. Por tanto, del mismo modo en que la palabra `case` identifica los puntos de partida posibles en la instrucción compuesta, la instrucción `break` determina puntos de terminación. Si se omiten las instrucciones `break`, se ejecutan todos los casos que siguen al valor `case` que tiene la correspondencia, incluyendo el caso `default`.

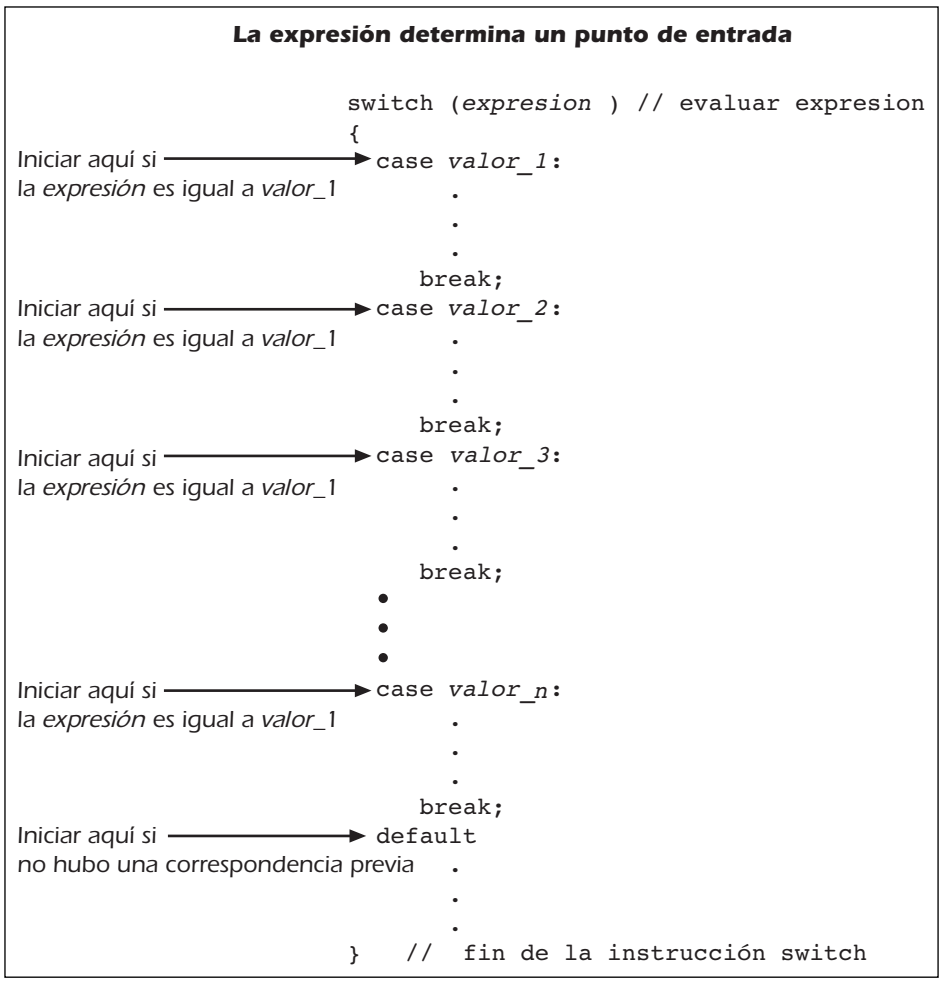


Figura 4.6 *La expresión determina un punto de entrada.*

Cuando escriba una instrucción `switch`, puede usar valores `case` múltiples para referirse al mismo conjunto de instrucciones; la etiqueta `default` es opcional. Por ejemplo, considere lo siguiente:

```
switch (numero)
{
  case 1:
    cout << "Que tenga una buena mañana\n";
    break;
  case 2:
    cout << "Que tenga un buen dia\n";
    break;
  case 3:
  case 4:
  case 5:
    cout << "Que tenga una buena tarde\n";
}
```

Si el valor almacenado en la variable número es 1, se despliega el mensaje `Que tenga una buena mañana`. Del mismo modo, si el valor de número es 2, se despliega el segundo mensaje. Por último, si el valor de número es 3 o 4 o 5, se despliega el último mensaje. En vista que la instrucción a ejecutarse en estos últimos tres casos es la misma, los casos para estos valores pueden "apilarse", como se muestra en el ejemplo. Además, en vista que no hay `default`, no se imprime ningún mensaje si el valor de número no es uno de los valores `case` enumerados. Aunque es una buena práctica de programación enlistar los valores `case` en orden ascendente, no lo requiere la instrucción `switch`. Una instrucción `switch` puede tener cualquier cantidad de valores `case`, en cualquier orden; sólo necesitan enlistarse los valores que se van a probar.

El programa 4.7 usa una instrucción switch para seleccionar la operación aritmética (adición, multiplicación o división) que se va a realizar con dos números dependiendo del valor de la variable `opselect`.

El programa 4.7 se ejecutó dos veces. El despliegue resultante identifica con claridad el caso seleccionado. Los resultados son

```
Por favor introduzca dos numeros: 12 3
Introduzca un codigo seleccionado:
          1 para adicion
          2 para multiplicacion
          3 para division : 2
El producto de los numeros introducidos es 36
```

y:

```
Por favor introduzca dos numeros: 12 3
Introduzca un código seleccionado:
          1 para adicion
          2 para multiplicacion
          3 para division : 3
El primer numero dividido entre el segundo es 4
```

Al revisar el programa 4.7, nótese la instrucción `break` en el último `case`. Aunque este `break` no es necesario, es una buena práctica terminar el último `case` en una instrucción `switch` con un `break`. Esto previene un posible error en el programa más tarde si después se agrega un case adicional a la instrucción `switch`. Con la adición de un `case` nuevo, el `break` entre `cases` se vuelve necesario; tener el `break` en su lugar le asegura que no olvidará incluirlo en el momento de la modificación.

Programa 4.7

```
#include <iostream>
using namespace std;

int main()
{
  int opselect;
  double fnum, snum;

  cout << "Por favor introduzca dos números: ";
  cin  >> fnum >> snum;
  cout << "Introduzca un código seleccionado: ";
  cout << "\n        1 para adicion";
  cout << "\n        2 para multiplicacion";
  cout << "\n        3 para division : ";
  cin  >> opselect;

  switch (opselect)
  {
    case 1:
      cout << "La suma de los numeros introducidos es " << fnum+snum;
      break;
    case 2:
      cout << "El producto de los numeros introducidos es " << fnum*snum;
      break;
    case 3:
      cout << "El primer numero dividido entre el segundo es " << fnum/snum;
      break;
  }     // end of switch

  cout << endl;

  return 0;
}
```

Debido a que los tipos de datos de carácter siempre son convertidos a números enteros en una expresión, también puede usarse una instrucción `switch` para "cambiar" con base en el valor de una expresión de carácter. Por ejemplo, suponiendo que `eleccion` es una variable de carácter, la siguiente instrucción `switch` es válida:

```
switch(eleccion)
{
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
```

```
    cout << "El caracter en eleccion es una vocal\n";
    break;
  default:
    cout << "El caracter en eleccion no es una vocal\n";
    break;    // este break es opcional
}    // fin de la instrucción switch
```

Ejercicios 4.4

1. Vuelva a escribir la siguiente cadena `if-else` usando una instrucción `switch`:

```
if (let_calif == 'A')
  cout << "La calificacion numerica esta entre 90 y 100\n";
else if (let_calif == 'B')
  cout << "La calificacion numerica esta entre 80 y 89.9\n";
else if (let_calif == 'C')
  cout << "La calificacion numerica esta entre 70 y 79.9\n";
else if (let_calif == 'D')
  cout << "Como va a explicar esta\n";
else
{
  cout << "Por supuesto que no tuve nada que ver con mi
           calificacion.\n";
  cout << "Debe ser culpa del profesor.\n";
}
```

2. Vuelva a escribir la siguiente cadena `if-else` usando una instrucción `switch`:

```
if (factor == 1)
  presion = 25.0;
else if (factor == 2)
  presion = 36.0;
else if (factor == 3)
 presion = 45.0;
else if (factor == 4) || (factor == 5) || (factor == 6)
  presion = 49.0;
```

3. Cada unidad de disco en un embarque de estos dispositivos tiene estampado un código del 1 al 4, el cual indica el fabricante de la unidad como sigue:

Código	Fabricante de la unidad de disco
1	3M Corporation
2	Maxell Corporation
3	Sony Corporation
4	Verbatim Corporation

Escriba un programa en C++ que acepte el número de código como una entrada y con base en el valor introducido despliegue el fabricante de la unidad de disco correcto.

4. Vuelva a escribir el programa 4.5 usando una instrucción `switch`.
5. Determine por qué la cadena `if-else` en el programa 4.6 no puede reemplazarse con una instrucción `switch`.
6. Vuelva a escribir el programa 4.7 usando una variable de carácter para el código seleccionado. (*Sugerencia:* revise la sección 3.4 si su programa no opera como pensaba que debería hacerlo.)

4.5 APLICACIONES

Dos usos principales de las instrucciones `if` de C++ son seleccionar rutas de procesamiento apropiadas y prevenir que los datos indeseables sean procesados. En esta sección se proporcionan ejemplos de ambos usos.

Aplicación 1: Validación de datos

Un uso importante de las instrucciones if de C++ es validar datos verificando los casos que son inválidos en forma clara. Por ejemplo, una fecha como 33/5/06 contiene un día que es obvio que es inválido. Del mismo modo, la división de cualquier número entre cero dentro de un programa, como 14/0, no debería permitirse. Estos dos ejemplos ilustran la necesidad de una técnica llamada **programación defensiva**, en la cual el programa incluye código para verificar datos impropios antes que se haga un intento de procesarlos más. La técnica de programación defensiva para verificar los datos introducidos por el usuario en busca de datos erróneos o irrazonables se conoce como **validación de datos de entrada**.

Considere el caso en el cual se escribe un programa en C++ para calcular la raíz cuadrada y el recíproco de un número introducido por un usuario. Antes de calcular la raíz cuadrada, validar que el número no sea negativo, y antes de calcular el recíproco, verificar que el número no sea cero.

Paso 1 Analizar el problema

El planteamiento del problema requiere que se acepte un solo número como entrada, se valide el número introducido y con base en la validación producir dos salidas posibles: si el número no es negativo se determina su raíz cuadrada, y si el número introducido no es cero se determina su recíproco.

Paso 2 Desarrollar una solución

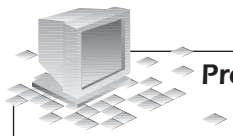
En vista que la raíz cuadrada de un número negativo no existe como un número real, y no puede tomarse el recíproco de cero, nuestro programa debe contener instrucciones de validación de datos de entrada para tamizar los datos introducidos por el usuario y evitar estos dos casos. El seudocódigo que describe el proceso requerido es:

Desplegar un mensaje con el propósito del programa
Aceptar un número introducido por un usuario

If el número es negativo
imprimir un mensaje indicando que no puede calcularse la raíz cuadrada
Else
calcular y desplegar la raíz cuadrada
Endif
If el número es cero entonces
imprimir un mensaje indicando que no puede obtenerse el recíproco
Else
calcular y desplegar el recíproco
Endif

Paso 3 Codificar la solución

El código C++ correspondiente a nuestra solución en seudocódigo se enumera en el programa 4.8.



Programa 4.8

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  double usenum;

  cout << "Este programa calcula la raiz cuadrada y\n"
       << "el reciproco (1/numero) de un numero\n"
       << "\nPor favor introduzca un numero: ";
  cin  >> usenum;
  if (usenum < 0.0)
    cout << "La raiz cuadrada de un numero negativo no existe.\n";
  else
    cout << "La raiz cuadrada de " << usenum
         << " es " << sqrt(usenum) << endl;
  if (usenum == 0.0)
    cout << "El reciproco de cero no existe.\n";
  else
    cout << "El reciproco de " << usenum
         << " es " <<  (1.0/usenum) << endl;

  return 0;
}
```

El programa 4.8 es un programa bastante sencillo que contiene dos instrucciones `if` separadas (no anidadas). La primera instrucción `if` verifica si se ha introducido un número negativo; si el número es negativo se despliega un mensaje que indica que no puede calcularse la raíz cuadrada de un número negativo, de lo contrario se calcula la raíz cuadrada. La segunda instrucción `if` comprueba si el número introducido es cero; si lo es se despliega un mensaje que indica que no puede obtenerse el recíproco de cero, de lo contrario se calcula el recíproco.

Paso 4 Probar y corregir el programa

Los valores de prueba deberán incluir un número positivo, y valores para los casos limitantes, como un valor de entrada negativo y un cero. A continuación se presenta la ejecución de prueba para dos de estos casos:

```
Este programa calcula la raiz cuadrada y
el reciproco (1/numero) de un numero

Por favor introduzca un numero: 5

La raiz cuadrada de 5 es 2.23607
El reciproco de 5 es 0.2
```

y

```
Este programa calcula la raiz cuadrada y
el reciproco (1/numero) de un numero

Por favor introduzca un numero: -6

La raiz cuadrada de un numero negativo no existe
El reciproco de -6 es -0.166667
```

Aplicación 2: Resolver ecuaciones cuadráticas

Una **ecuación cuadrática** es una ecuación que tiene la forma $ax^2 + bx + c = 0$ o que puede manipularse en forma algebraica en esta forma. En esta ecuación x es la variable desconocida, y a, b y c son constantes conocidas. Aunque las constantes b y c pueden ser cualquier número, incluyendo cero, el valor de la constante a no puede ser cero (si a es cero, la ecuación se volvería una **ecuación lineal** en x). Son ejemplos de ecuaciones cuadráticas

$$5x^2 + 6x + 2 = 0$$
$$x^2 - 7x + 20 = 0$$
$$34x^2 + 16 = 0$$

En la primera ecuación $a = 5$, $b = 6$ y $c = 2$; en la segunda ecuación $a = 1$, $b = -7$ y $c = 20$; y en la tercera ecuación $a = 34$, $b = 0$ y $c = 16$.

Las raíces reales de una ecuación cuadrática pueden calcularse usando la fórmula cuadrática como:

$$\text{raíz 1} = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

y

$$\text{raíz 2} = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Usando estas ecuaciones escribiremos un programa en C++ para resolver las raíces de una ecuación cuadrática.

Paso 1 Analizar el problema

El problema requiere que se acepten tres entradas: los coeficientes *a*, *b* y *c* de una ecuación cuadrática, y calcular las raíces de la ecuación usando las fórmulas dadas.

Paso 2 Desarrollar una solución

Un primer intento de solución sería usar los valores de *a*, *b* y *c* introducidos por el usuario para calcular en forma directa un valor para cada una de las raíces. Por tanto, nuestra primera solución sería

```
Desplegar un mensaje con el propósito del programa
Aceptar valores introducidos por el usuario para a, b y c
Calcular las dos raíces
Desplegar los valores calculadas de las raíces
```

Sin embargo, esta solución debe ser definada para tomar en cuenta las posibles condiciones de entrada. Por ejemplo, si un usuario introduce un valor de *D* para *a* y *b*, la ecuación no es ni cuadrática ni lineal y no tiene solución (se hace referencia como un caso degenerado). Otra posibilidad es que el usuario proporcione un valor no cero para *b* por lo que *a* es cero. En este caso la ecuación es un uno lineal con una solución $-c / b$. Una tercera posibilidad es que el valor del término $b^2 - 4ac$, el cual es llamado **diseniminante**, sea negativo. Entonces la raíz cuadrada de un número negativo no se tomará en cuenta, este caso no tendrá raíces verdaderás. Por último, cuando el diseniminante es cero ambas raíces son las mismas (se hace referencia a ello como un caso de repetición de raíces).

Tomando en cuenta estos cuatro casos limitantes, una solución refinada para la correcta determinada de las raíces de la ecuación cuadríatica es expresada por el siguiente seudo código:

```
Desplegar un mensaje con el propósito del programa
Aceptan valores introduccidos por el usuario para a, b y c
If a = 0 y b = 0 entonces
    desplegar un mensaje diciendo que la ecuación no tiene solución
Else if a = cero entonces
    calcular la raíz individual igual a –c/b
    desplegar la raíz individual
Else
    calcular el discriminante
```

```
    If el discriminante > 0 entonces
        resolver ambas raíces usando las fórmulas dadas
        desplegar las dos raíces
    Else if el discriminante < 0 entonces
        desplegar un mensaje de que no hay raíces reales
    Else
        calcular la raíz repetida igual a –b/(2a)
        desplegar la raíz repetida
    Endif
Endif
```

Nótese en el seudocódigo que se han usado instrucciones `if-else` anidadas. La instrucción `if-else` exterior se usa para validar los coeficientes introducidos y determinar que se tiene una ecuación cuadrática válida. La instrucción `if-else` interior se usa entonces para determinar si la ecuación tiene dos raíces reales (discriminante > 0), dos raíces imaginarias (discriminante < 0) o raíces repetidas (discriminante = 0).

Paso 3 Codificar la solución

El código C++ equivalente correspondiente a nuestra solución en seudocódigo se enlista en el programa 4.9.

Paso 4 Probar y corregir el programa

Los valores de prueba deberán incluir valores para a, b y c que produzcan dos raíces reales, además de valores limitantes para a y b que produzcan una ecuación lineal ($a = 0$, $b \neq 0$), una ecuación degenerada ($a = 0$, $b = 0$), y un discriminante negativo y cero. A continuación se muestran dos de estas ejecuciones de prueba del programa 4.9:

```
Este programa calcula las raices de una
   ecuacion cuadratica de la forma
            2
          ax + bx + c = 0
Por favor introduzca valores para a, b y c: 1 2 -35

Las dos raices reales son 5 y -7
```

y

```
Este programa calcula las raices de una
   ecuacion cuadratica de la forma
            2
          ax + bx + c = 0

Por favor introduzca valores para a, b y c: 0 0 16

La ecuacion es degenerada y no tiene raices.
```

Programa 4.9

```
#include <iostream>
#include <cmath>
using namespace std;

// este programa calcula las raices de una ecuacion cuadratica
int main()
{
  double a, b, c, disc, raiz1, raiz2;

  cout << "Este programa calcula las raices de una\n";
  cout << "   ecuacion cuadratica de la forma\n";
  cout << "             2\n";
  cout << "          ax + bx + c = 0\n\n";
  cout << "Por favor introduzca valores para a, b y c: ";
  cin >>  a >>  b >> c;
  if ( a == 0.0 && b == 0.0)
    cout << "La ecuacion es degenerada y no tiene raices.\n";
  else if (a == 0.0)
    cout << "La ecuacion tiene la raiz unica x = "
       << -c/b << endl;
  else
  {
    disc = pow(b,2.0) - 4 * a * c;    // calcula el discriminante
    if (disc > 0.0)
    {
      disc = sqrt(disc);
      root1 = (-b + disc) / (2 * a);
      root2 = (-b - disc) / (2 * a);
      cout << "Las dos raices reales son "
   << root1 << " y " << raiz2 << endl;
    }
    else if (disc < 0.0)
      cout << "Ambas raices son imaginarias.\n";
    else
      cout << "Ambas raices son iguales a " << -b / (2 * a) << endl;
  }

  return 0;
}
```

La primera ejecución resuelve la ecuación cuadrática $x^2 + 2x - 35 = 0$, la cual tiene las raíces reales $x = 5$ y $x = -7$. Los datos de entrada para la segunda ejecución producen la ecuación $0x^2 + 0x + 16 = 0$. Debido a que esto degenera en la imposibilidad matemática de $16 = 0$, el programa identifica en forma correcta ésta como una ecuación degenerada. Dejamos como ejercicio crear datos de prueba para los otros casos limitantes verificados por el programa.

Ejercicios 4.5

1. **a.** Escriba un programa que acepte dos números reales de un usuario y un código seleccionado. Si el código seleccionado introducido es 1, haga que el programa sume los dos números introducidos con anterioridad y despliegue el resultado; si el código seleccionado es 2, los números deberán multiplicarse, y si el código seleccionado es 3, el primer número deberá ser dividido entre el segundo número.
   b. Determine qué hace el programa escrito en el ejercicio 1a cuando los números introducidos son 3 y 0, y el código seleccionado es 3.
   c. Modifique el programa escrito en el ejercicio 1a de modo que no se permita la división entre 0 y se despliegue un mensaje apropiado cuando se intente dicha división.

2. **a.** Escriba un programa para desplegar los dos indicadores siguientes:

```
Introduzca un mes (use 1 para Ene, etc.):
Introduzca un dia del mes:
```

   Haga que su programa acepte y almacene un número en la variable `mes` en respuesta al primer indicador, y acepte y almacene un número en la variable `dia` en respuesta al segundo indicador. Si el mes introducido no está entre 1 y 12 inclusive, imprima un mensaje informando al usuario que se ha introducido un mes inválido. Si el día introducido no está entre 1 y 31, imprima un mensaje informando al usuario que se ha introducido un día inválido.
   b. ¿Qué hará su programa si el usuario introduce un número con un punto decimal para el mes? ¿Cómo puede asegurar que sus instrucciones `if` comprueben que es un número entero?
   c. En un año que no es bisiesto, febrero tiene 28 días, los meses de enero, marzo, mayo, julio, agosto, octubre y diciembre tienen 31 días y todos los demás meses tienen 30 días. Usando esta información, modifique el programa escrito en el ejercicio 2a para desplegar un mensaje cuando se introduzca un día inválido para un mes introducido por un usuario. Para este programa ignore los años bisiestos.

3. **a.** El cuadrante en el que reside una línea trazada desde el origen es determinado por el ángulo que forma la línea con el eje `x` positivo como sigue:

Ángulo desde el eje *x* positivo	Cuadrante
Entre 0 y 90 grados	I
Entre 90 y 180 grados	II
Entre 180 y 270 grados	III
Entre 270 y 360 grados	IV

Usando esta información, escriba un programa en C++ que acepte el ángulo de la línea como una entrada del usuario y determine y despliegue el cuadrante apropiado a los datos introducidos. (NOTA: Si el ángulo tiene exactamente 0, 90, 180 o 270 grados, la línea correspondiente no reside en ningún cuadrante sino que se encuentra en un eje.)

b. Modifique el programa escrito para el ejercicio 3a de modo que se despliegue un mensaje que identifique un ángulo de cero grados como el eje *x* positivo, un ángulo de 90 grados como el eje y positivo, un ángulo de 180 grados como el eje *x* negativo y un ángulo de 270 grados como el eje *y* negativo.

4. Todos los años que se dividen exactamente entre 400 o que son divisibles exactamente entre cuatro y no son divisibles exactamente entre 100 son años bisiestos. Por ejemplo, en vista que 1600 es divisible exactamente entre 400, el año 1600 fue un año bisiesto. Del mismo modo, en vista que 1988 es divisible exactamente entre cuatro pero no entre 100, el año 1988 también fue un año bisiesto. Usando esta información, escriba un programa en C++ que acepte el año como una entrada del usuario, determine si el año es un año bisiesto y despliegue un mensaje apropiado que le indique al usuario si el año introducido es un año bisiesto o no.

5. Con base en el año del modelo y el peso de un automóvil el estado de Nueva Jersey determina la clase del vehículo y la tarifa de registro que le corresponde usando la siguiente tabla:

Año del modelo	Peso	Clase de peso	Tarifa de registro
1970 o anterior	Menos de 2700 lbs	1	$16.50
	2700 a 3800 lbs	2	25.50
	Más de 3800 lbs	3	46.50
1971 a 1979	Menos de 2700 lbs	4	27.00
	2700 a 3800 lbs	5	30.50
	Más de 3800 lbs	6	52.50
1980 o posterior	Menos de 3500 lbs	7	19.50
	3500 lbs o más	8	52.50

Usando esta información, escriba un programa en C++ que acepte el año y el peso de un automóvil y determine y despliegue la clase y la tarifa de registro para el automóvil.

6. Modifique el programa 4.9 de modo que se calculen y desplieguen las raíces imaginarias cuando el discriminante es negativo. Para este caso las dos raíces de la ecuación son:

$$x_1 = \frac{-b}{2a} + \frac{\text{sqrt}[-(b^2 - 4ac)]}{2a} i$$

y

$$x_2 = \frac{-b}{2a} - \frac{\text{sqrt}[-(b^2 - 4ac)]}{2a} i$$

donde *i* es el símbolo del número imaginario para la raíz cuadrada de –1. (*Sugerencia*: Calcule las partes real e imaginaria de cada raíz por separado.)

7. En el juego del 21, el valor de las cartas del 2 al 10 es el que tienen impreso, sin importar de qué palo sean. Las cartas de personajes (jota, reina y rey) se cuentan como 10, y el as se cuenta como 1 u 11, dependiendo de la suma de todas las cartas en una mano. El as se cuenta como 11 sólo si el valor total resultante de todas las cartas en una mano no excede de 21, de lo contrario se cuenta como 1. Usando esta información, escriba un programa en C++ que acepte los valores de tres cartas como entradas (un 1 correspondiente a un as, un 2 correspondiente a un dos, etc.), calcule el valor total de la mano en forma apropiada y despliegue el valor de las tres cartas con un mensaje impreso.

4.6 ERRORES COMUNES DE PROGRAMACIÓN

Tres errores de programación son comunes en las instrucciones de selección de C++.

1. Usar el operador de asignación, =, en lugar del operador relacional, ==. Esto puede causar una enorme cantidad de frustración debido a que cualquier expresión puede ser probada por una instrucción `if-else`. Por ejemplo, la instrucción

```
if (opselect = 2)
   cout << "Feliz Cumpleaños";
else
   cout << "Buen Dia";
```

   siempre produce que se imprima el mensaje `Feliz Cumpleaños`, sin importar el valor inicial en la variable `opselect`. La razón para esto es que la expresión de asignación `opselect = 2` tiene un valor de 2, el cual se considera un valor verdadero en C++. La expresión correcta para determinar el valor en `opselect` es `opselect == 2`.

2. Permitir que la instrucción `if-else` aparente seleccionar una opción incorrecta. En este problema de depuración típico, el programador se concentra en forma errónea en la condición probada como la fuente del problema. Por ejemplo, suponga que la siguiente instrucción `if-else` es parte de su programa:

```
if (clave == 'F')
{
  contemp = (5.0/9.0) * (intemp - 32.0);
  cout << "Se efectuo la conversion a grados Celsius";
}
else
{
  contemp = (9.0/5.0) * intemp + 32.0;
  cout << "Se efectuo la conversion a grados Fahrenheit";
}
```

Esta instrucción siempre desplegará Se efectuo la `conversion` a grados `Celsius` cuando la variable `clave` contenga una `F`. Por consiguiente, si este mensaje se despliega cuando usted cree que `clave` no contiene una `F`, requerirá investigar el valor de `clave`. Como regla general, siempre que una instrucción de selección no actúa como piensa que debería, pruebe sus suposiciones acerca de los valores asignados a las variables probadas desplegando sus valores. Si se despliega un valor no anticipado, al menos habrá aislado la fuente del problema en las variables mismas, en lugar de buscarla en la estructura de la instrucción `if-else`. A partir de ahí tendrá que determinar dónde y cómo se obtuvo el valor incorrecto.

3. Usar instrucciones `if` anidadas sin incluir llaves para indicar la estructura deseada. Sin llaves, el compilador empareja por omisión los `else` con los `if` sin par más cercanos, lo cual a veces destruye la intención original de la instrucción de selección. Para evitar este problema y crear código que se adapte con facilidad al cambio, es útil escribir todas las instrucciones `if-else` como instrucciones compuestas en la forma

```
if (expresion)
{
  una o mas instrucciones entran aqui
}
else
{
  una o mas instrucciones entran aquí
}
```

Al utilizar esta forma, la integridad y la intención originales de la instrucción `if` se mantienen sin importar cuántas instrucciones se agreguen después.

4.7 RESUMEN DEL CAPÍTULO

1. Las expresiones relacionales, las cuales también se llaman ***condiciones***, se usan para comparar operandos. Si una expresión relacional es verdadera, el valor de la expresión es el entero 1. Si la expresión relacional es falsa, tiene un valor entero de 0. Las expresiones relacionales se crean usando los siguientes operadores relacionales.

Operador relacional	Significado	Ejemplo
<	Menor que	`edad < 30`
>	Mayor que	`altura > 6.2`
<=	Menor que o igual a	`gravable <= 20000`
>=	Mayor que o igual a	`temp >= 98.6`
==	Igual a	`calificación == 100`
!=	No es igual a	`numero != 250`

2. Pueden construirse condiciones más complejas a partir de las expresiones relacionales usando los operadores lógicos de C++, `&&` (AND), `||` (OR) y `!` (NOT).

3. Se usa una instrucción `if-else` para seleccionar entre dos instrucciones alternativas con base en el valor de una expresión. Aunque las expresiones relacionales se usan por lo general para la expresión probada, puede usarse cualquier expresión válida. Al probar una expresión, las instrucciones `if-else` interpretan un valor diferente de cero como verdadero y un valor de cero como falso. La forma general de una instrucción `if-else` es:

```
if (expresion)
  instruccion1;
else
  instruccion2;
```

   Ésta es una instrucción de selección bidireccional. Si la expresión tiene un valor diferente de cero es considerada como verdadera y se ejecuta `instruccion1`; de lo contrario se ejecuta `instruccion2`.

4. Una instrucción `if-else` puede contener otras instrucciones `if-else`. En ausencia de llaves, cada `else` se asocia con el `if` sin par precedente más cercano.

5. La cadena `if-else` es una instrucción de selección de vía múltiple que tiene la forma general

```
if (expresion_1)
  instruccion_1;
else if (expresion_2)
  instruccion_2;
else if (expresion_3)
  instruccion_3;
      .
      .
      .
else if (expresion_m)
  instruccion_m;
else
  instruccion_n;
```

   Cada expresión es evaluada en el orden en que aparece en la cadena. Una vez que una expresión es verdadera (tiene un valor diferente de cero), sólo se ejecuta la instrucción entre esa expresión y el siguiente `else if` o `else`, y no se prueban más expresiones. El `else` final es opcional, y la instrucción correspondiente al `else` final sólo se ejecuta si ninguna de las expresiones anteriores es verdadera.

6. Una instrucción compuesta consiste en cualquier cantidad de instrucciones individuales encerradas dentro del par de llaves `{` y `}`. Las instrucciones compuestas son tratadas como una sola unidad y pueden usarse en cualquier parte en que se use una instrucción individual.

7. La instrucción `switch` es una instrucción de selección de vía múltiple. La forma general de una instrucción switch es

```
switch (expresion)
{    // inicio de la instruccion compuesta
  case valor_1:  <-------------- termina con dos puntos
    instruccion1;
    instruccion2;
        .
        .
    break;
  case valor_2:  <-------------- termina con dos puntos
    instruccionm;
    instruccionn;
        .
        .
    break;
   .
   .
  case valor_n:  <-------------- termina con dos puntos
    instruccionw;
    instruccionx;
        .
        .
    break;
  default:       <-------------- termina con dos puntos
    instruccionaa;
    instruccionbb;
        .
        .
}    // fin de switch y de la instruccion compuesta
```

Para esta instrucción el valor de una expresión es comprada con un número entero o con una constante o con expresiones constantes. La ejecución del programa se transfiere al primer caso de correspondencia y continúa hasta el final de la instrucción `switch` a menos que encuentre una instrucción `break` opcional. Los casos en una instrucción `switch` pueden aparecer en cualquier orden y puede incluirse un caso `default` opcional. El caso `default` se ejecuta si ninguno de los otros casos encuentra una correspondencia.

4.8 APÉNDICE DEL CAPÍTULO: UN ACERCAMIENTO MÁS A FONDO A LA PRUEBA EN PROGRAMACIÓN

En teoría, un conjunto extenso de ejecuciones de prueba revelaría todos los errores posibles del programa y aseguraría que éste funcionará en forma correcta para todas y cada una de las combinaciones de datos de entrada y datos calculados. En la práctica esto requiere verificar todas las combinaciones posibles de ejecución de instrucciones. Debido al tiempo y esfuerzo requeridos, ésta es una meta imposible excepto para programas extremadamente simples. Veamos por qué es así. Considere el programa 4.10.

Programa 4.10

```
#include <iostream>
using namespace std;

int main()
{
  int num;

  cout << "Introduzca un número: ";
  cin  >> num;
  if (num == 5)
    cout << "¡Loteria!\n";
  else
    cout << "¡Lotoria!\n";

  return 0;
}
```

El programa 4.10 tiene dos rutas que pueden recorrerse conforme el programa progresa desde su llave de apertura hasta su llave de cierre. La primera ruta, la cual se ejecuta cuando el número introducido es 5, está en la secuencia

```
cout << "Introduzca un número";
cin  >> num;
cout << "¡Loteria!\n";
```

La segunda ruta, la cual se ejecuta siempre que se introduce cualquier número excepto 5, incluye la secuencia de instrucciones

```
cout << "Introduzca un número";
cin  >> num;
cout << "¡Lotoria!\n";
```

Probar cada ruta posible en el programa 4.10 requiere dos ejecuciones del programa, con una selección juiciosa de los datos de entrada de prueba para asegurar que se emplean ambas rutas de la instrucción `if`. La adición de una instrucción `if` más en el programa incrementa el número de rutas de ejecución posibles por un factor de dos y requiere cuatro ejecuciones (2^2) del programa para una prueba completa. Del mismo modo, dos instrucciones `if` adicionales incrementan el número de rutas por un factor de cuatro y requieren ocho ejecuciones (2^3) para una prueba completa y tres instrucciones `if` adicionales producirían un programa que requiere dieciséis (2^4) ejecuciones de prueba.

Ahora considere un programa de aplicación de tamaño modesto que consta sólo de diez módulos, en el que cada módulo contiene cinco instrucciones `if`. Suponiendo que los módulos siempre son invocados en la misma secuencia, hay 32 rutas posibles a través de cada módulo (2 elevado a la quinta potencia) y más de 1 000 000 000 000 000 (2 elevado a la quincuagésima potencia) de rutas posibles a través del programa completo (todos los módulos ejecutados en secuencia). El tiempo necesario para crear datos de prueba individuales para emplear cada ruta y el tiempo de ejecución real en la computadora requerido para verificar cada ruta hace que la prueba completa de un programa así sea imposible de lograr.

La incapacidad para probar por completo todas las combinaciones de secuencias de ejecución de instrucciones ha conducido al dicho de programación de que "no hay programa libre de errores". También ha conducido a percatarse de que cualquier prueba que se haga deberá pensarse bien para maximizar la posibilidad de localizar errores. Como mínimo, los datos de prueba deberán incluir valores apropiados para los valores de entrada, valores de entrada ilegales que el programa deberá rechazar y valores limitantes que son verificados por las instrucciones de selección dentro del programa.

Consideración de opciones de carrera

Ingeniería civil

El campo de la ingeniería civil se interesa sobre todo en las estructuras y sistemas a gran escala usados por una comunidad. Un ingeniero civil diseña, construye y opera puentes, presas, túneles, edificios, aeropuertos, carreteras y otras obras públicas a gran escala. Los ingenieros civiles también son responsables de los efectos que tienen estos sistemas a gran escala en la sociedad y en el ambiente. Por tanto, los ingenieros civiles están implicados en los recursos hidráulicos, el control de inundaciones, la eliminación de desperdicios y la planeación urbana general. El campo puede subdividirse en tres categorías.

1. Estructuras. Diseño, construcción y operación de edificios a gran escala como presas, edificaciones y carreteras. Las propiedades de los materiales, la geología, la mecánica de suelos, y la estática y dinámica son elementos importantes en su educación. Por ejemplo, qué tan alto puede construirse un edificio antes que lo venza su propio peso es una cuestión que incluye todos estos temas.
2. Planeación urbana. La planeación, diseño y construcción de sistemas de transporte (carreteras, ferrocarriles, desarrollo fluvial, aeropuertos) y el uso general de la tierra. La topografía y la cartografía son habilidades necesarias.
3. Sanidad. Tratamiento de desechos, suministro de agua y sistemas de drenaje. La mecánica de fluidos, hidrología, control de la contaminación, irrigación y economía son consideraciones importantes.

CAPÍTULO 5
Instrucciones de repetición

TEMAS

5.1 ESTRUCTURAS BÁSICAS DEL CICLO
- CICLOS DE PRUEBA PRELIMINAR Y PRUEBA POSTERIOR
- CICLOS DE CUENTA FIJA FRENTE A LOS DE CONDICIÓN VARIABLE

5.2 CICLOS `while`

5.3 CICLOS `while` INTERACTIVOS
- CENTINELAS
- INSTRUCCIONES `break` Y `continue`
- LA INSTRUCCIÓN NULA

5.4 CICLOS `for`

5.5 TÉCNICAS DE PROGRAMACIÓN CON CICLOS
- TÉCNICA 1: ENTRADA INTERACTIVA DENTRO DE UN CICLO
- TÉCNICA 2: SELECCIÓN DENTRO DE UN CICLO
- TÉCNICA 3: EVALUACIÓN DE FUNCIONES DE UNA VARIABLE
- TÉCNICA 4: CONTROL INTERACTIVO DE UN CICLO

5.6 CICLOS ANIDADOS

5.7 CICLOS `do while`
- VERIFICACIONES DE VALIDEZ

5.8 ERRORES COMUNES DE PROGRAMACIÓN

5.9 RESUMEN DEL CAPÍTULO
- CONSIDERACIÓN DE OPCIONES DE CARRERA: INGENIERÍA INDUSTRIAL

Los programas examinados hasta ahora han ilustrado los conceptos de programación implicados en las capacidades de entrada, salida, asignación y selección. En este punto ya debe tener suficiente experiencia para sentirse cómodo con estos conceptos y la mecánica de ponerlos en práctica usando C++. Sin embargo, muchos problemas requieren una capacidad de repetición en la cual el mismo cálculo o secuencia de instrucciones se repita, una y otra vez, usando diferentes conjuntos de datos. Los ejemplos de dicha repetición incluyen la verificación continua de las entradas de datos del usuario hasta que se introduce una entrada aceptable, como una contraseña válida; contar y acumular

totales corrientes; y la aceptación constante de datos de entrada y el recálculo de valores de salida que sólo se detiene al introducir un valor centinela.

Este capítulo explora los diferentes métodos que usan los programadores para construir secciones repetitivas de código y cómo pueden ponerse en práctica en C++. Por lo común, a una sección de código que se repite se le conoce como ***ciclo****, porque después que se ejecuta la última instrucción en el código el programa se ramifica, o regresa, a la primera instrucción y comienza otra repetición a través del código. Cada repetición se conoce también como una* ***iteración*** *o* ***paso a través del ciclo****.*

5.1 ESTRUCTURAS BÁSICAS DEL CICLO

El poder real de un programa se demuestra cuando debe hacerse una y otra vez el mismo tipo de operación. Por ejemplo, considere el programa 3.17 en la sección 3.6, donde el mismo conjunto de instrucciones se repite tres veces. Volver a escribir este mismo conjunto de instrucciones es tedioso, consume tiempo y está sujeto a errores. Por supuesto que sería conveniente si pudiéramos escribir esas instrucciones repetitivas sólo una vez y luego implementar un método para informar al programa que repita la ejecución de las instrucciones tres veces. Dicho método está disponible usando secciones de código repetitivas.

Construir una sección de código repetitiva requiere que estén presentes cuatro elementos. El primer elemento necesario es una instrucción de repetición. Esta **instrucción de repetición** define los límites que contienen la sección de código repetitiva y controla si el código se ejecutará o no. En general, hay tres formas diferentes de instrucciones de repetición, todas las cuales son proporcionadas en C++:

1. `while`
2. `for`
3. `do while`

Cada una de estas instrucciones requiere una condición que debe evaluarse, la cual es el segundo elemento requerido para construir secciones de código repetitivas. Las condiciones válidas son idénticas a las usadas en las instrucciones de selección. Si la condición es verdadera, el código es ejecutado; de lo contrario, no lo es.

El tercer elemento requerido es una instrucción que establece la condición al inicio. Esta instrucción debe colocarse siempre antes que la condición sea evaluada por primera vez para asegurar la ejecución correcta del ciclo.

Por último, debe haber una instrucción dentro de la sección de código repetitiva que permita que la condición se vuelva falsa. Esto es necesario para asegurar que, en algún punto, se detengan las repeticiones.

Ciclos de prueba preliminar y prueba posterior

La condición que se está probando puede evaluarse al principio o al final de la sección de código que se repite. La figura 5.1 ilustra el caso donde la prueba ocurre al principio del ciclo. Este tipo de ciclo se conoce como **ciclo de prueba preliminar** porque la condición se prueba antes que se ejecuten cualesquiera instrucciones dentro del ciclo. Si la condición es verdadera, se ejecutan las instrucciones ejecutables dentro del ciclo. Si el valor inicial de la condición es falso, las instrucciones ejecutables dentro del ciclo nunca se ejecutan y el control se transfiere a la primera instrucción que se halla después del ciclo. Para evitar repeticiones infinitas, la condición debe actualizarse dentro del ciclo. Los ciclos de prueba preliminar también se conocen como **ciclos controlados en la entrada**. Las estructuras de ciclo `while` y `for` son ejemplos de estos ciclos.

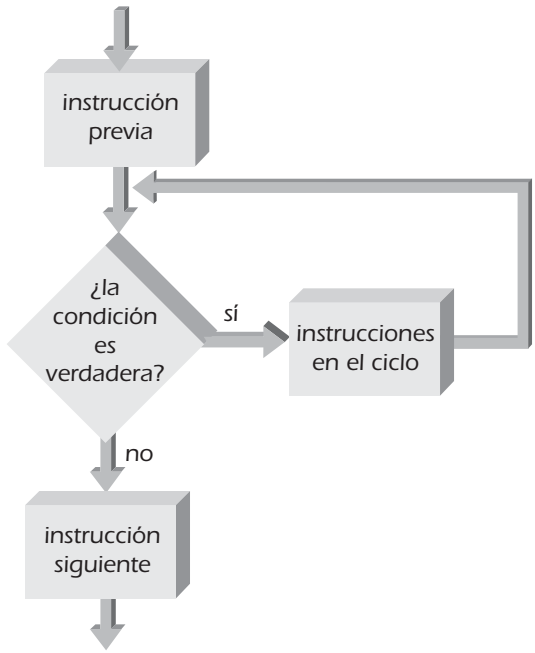


Figura 5.1 Un ciclo de prueba preliminar.

Un ciclo que evalúa una condición al final de la sección de código repetitiva, como se ilustra en la figura 5.2, se conoce como un **ciclo de prueba posterior** o **ciclo controlado en la salida**. Estos ciclos siempre ejecutan las instrucciones del ciclo al menos una vez antes que la condición se pruebe. En vista que las instrucciones ejecutables dentro del ciclo se ejecutan de manera continua hasta que la condición se vuelve falsa, siempre debe haber una instrucción dentro del ciclo que actualice la condición y permita que se vuelva falsa. El constructo `do while` es un ejemplo de un ciclo de prueba posterior.

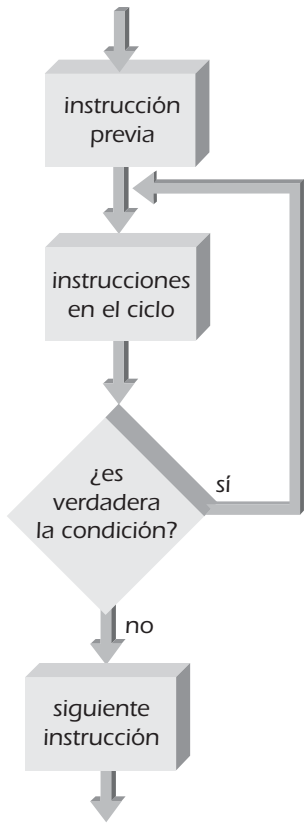


Figura 5.2 *Un ciclo de prueba posterior.*

Ciclos de cuenta fija frente a los de condición variable

Además del lugar donde se prueba la condición (prueba preliminar o prueba posterior), las secciones de código repetitivas también se clasifican según el tipo de condición que se prueba. En un **ciclo de cuenta fija**, la condición se usa para dar seguimiento al número de repeticiones que han ocurrido. Por ejemplo, podríamos desear producir una tabla de 10 números, incluyendo sus cuadrados y cubos, o un diseño fijo como

```
**************************
**************************
**************************
**************************
```

En cada uno de estos casos, se ejecuta un número fijo de cálculos o se imprime un número fijo de líneas, punto en el cual la sección de código repetitiva finaliza. Todas las instrucciones de repetición de C++ pueden utilizarse para producir ciclos de cuenta fija.

En muchas situaciones no se conoce con anticipación el número exacto de repeticiones o los elementos son demasiado numerosos para contarlos con antelación. Por ejemplo, cuando se introduce una cantidad grande de datos experimentales podríamos no desear tomar el tiempo para contar el número de elementos de datos real que se van a introducir. En casos como éste se usa un ciclo de condición variable. En un **ciclo de condición variable** la condi-

ción probada no depende de que se alcance una cuenta, sino más bien de una variable que puede cambiar de manera interactiva con cada paso a través del ciclo. Cuando se encuentra un valor especificado, sin importar cuántas iteraciones han ocurrido, las repeticiones se detienen. Todas las instrucciones de repetición de C++ pueden usarse para crear ciclos de condición variable.[1] En este capítulo se encontrarán ejemplos de ciclos de cuenta fija y de condición variable.

Ejercicios 5.1

1. Enumere las tres instrucciones de repetición que se proporcionan en C++.
2. Enumere los cuatro elementos que deben estar presentes en una instrucción de repetición.
3. **a.** ¿Qué es un ciclo controlado en la entrada?
   b. ¿Cuál de las instrucciones de repetición de C++ produce ciclos controlados en la entrada?
4. **a.** ¿Qué es un ciclo controlado en la salida?
   b. ¿Cuál de las instrucciones de repetición de C++ produce ciclos controlados en la salida?
5. **a.** ¿Cuál es la diferencia entre un ciclo de prueba preliminar y uno de prueba posterior?
   b. Si la condición que se está probando en un ciclo de prueba preliminar es falsa para comenzar, ¿cuántas veces se ejecutarán las instrucciones internas del ciclo?
   c. Si la condición que se está probando en un ciclo de prueba posterior es falsa para comenzar, ¿cuántas veces se ejecutarán las instrucciones internas del ciclo?
6. ¿Cuál es la diferencia entre un ciclo de cuenta fija y uno de condición variable?

5.2 Ciclos `while`

En C++, un **ciclo `while`** se construye usando una instrucción `while`. La sintaxis de esta instrucción es

```
while (expresión)
   instrucción;
```

La *`expresión`* contenida dentro del paréntesis es la condición probada para determinar si se ejecuta la *`instrucción`* que sigue al paréntesis. La expresión es evaluada exactamente en la misma manera que la contenida en una instrucción `if-else`; la diferencia está en cómo se usa la expresión. Como se ha visto, cuando la expresión es verdadera (tiene un valor diferente de cero) en una instrucción `if-else`, la instrucción que sigue a la expresión se ejecuta una vez. En una instrucción `while`, la instrucción que sigue a la expresión se ejecuta en forma repetida hasta que la expresión reconozca un valor diferente de cero. Considerando

[1] En esto, C y C++ difieren de los lenguajes de alto nivel anteriores, en los que la instrucción `for` (la cual se ponía en práctica usando una instrucción `DO` en FORTRAN) sólo podía usarse para producir ciclos de cuenta fija. La instrucción `for` de C++, como se verá en breve, es casi intercambiable con su instrucción `while`.

sólo la expresión y la instrucción que sigue al paréntesis, el proceso usado por la computadora al evaluar una instrucción `while` es

1. ***Probar la expresión***
2. ***Si la expresión tiene un valor diferente de cero (verdadero)***
   a. ejecutar la instrucción que sigue al paréntesis
   b. regresar al paso 1
   de lo contrario
   salir de la instrucción `while` ***y ejecutar la siguiente instrucción ejecutable que sigue a la instrucción*** `while`

Hay que observar que el paso 2b obliga a que el control del programa se transfiera de nuevo al paso 1. Esta transferencia del control de vuelta al inicio de una instrucción `while` a fin de reevaluar la expresión es lo que forma el ciclo del programa. La instrucción `while` literalmente se enrrolla en sí misma para volver a verificar la expresión hasta que se evalúe en cero (se vuelva falsa). Esto naturalmente significa que en alguna parte en el ciclo debe estipularse una disposición que permita que se altere el valor de la expresión probada. Como se verá, esto es así.

Este proceso de repetición producido por una instrucción `while` se ilustra en la figura 5.3. Se usa una forma de diamante para mostrar los dos puntos de entrada y los dos puntos de salida requeridos en la parte de decisión de la instrucción `while`.

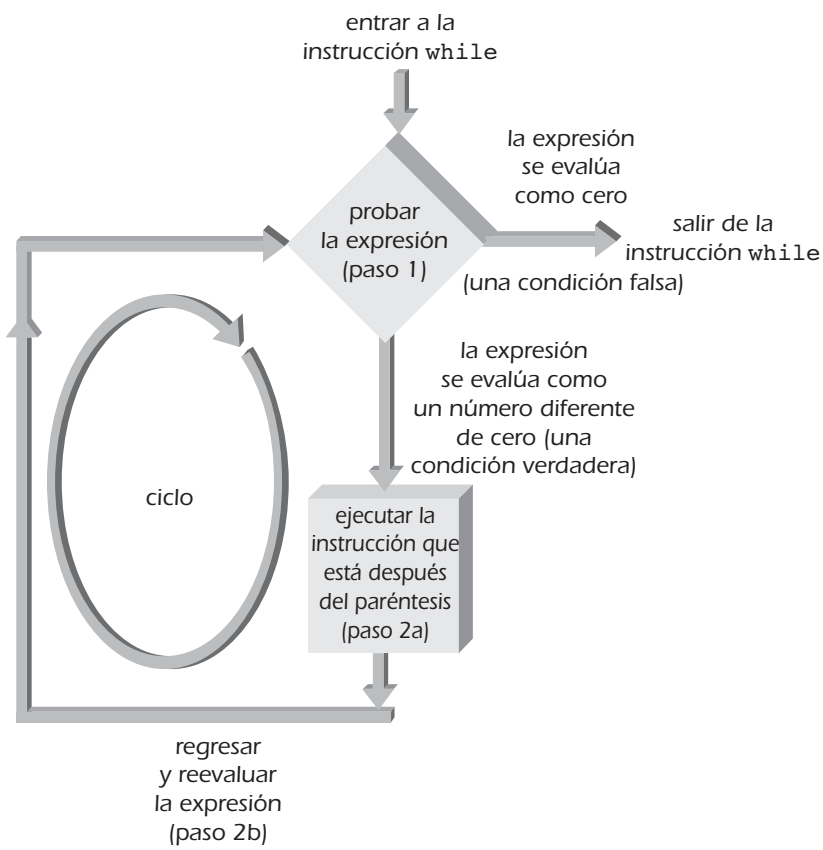


Figura 5.3 *Anatomía de un ciclo* `while`.

Para hacer esto un poco más tangible, considere la expresión relacional `cuenta <= 10` y la instrucción `cout << cuenta;`. Uando estas dos instrucciones, se puede escribir la siguiente instrucción `while` válida:

```
while (cuenta <= 10)
  cout << cuenta;
```

Aunque la instrucción anterior es válida, el lector alerta se percatará que se ha creado una situación en la que el objeto `cout` es invocado de manera indefinida (hasta que detengamos el programa) o no es invocado en absoluto. Veamos por qué sucede esto.

Si `cuenta` tiene un valor menor que o igual a 10 cuando la expresión se evalúa por primera vez, se ejecuta la instrucción `cout`. Entonces la instrucción `while` regresa automáticamente y vuelve a probar la expresión. En vista que no se ha cambiado el valor almacenado en cuenta, la expresión aún es verdadera y se hace otra llamada a `cout`. Este proceso continúa de manera indefinida, hasta que el programa que contiene esta instrucción sea detenido por el usuario. Sin embargo, si `cuenta` inicia con un valor mayor que 10, la expresión es falsa para comenzar y el objeto `cout` nunca se usa.

¿Cómo establecemos un valor inicial en `cuenta` para controlar lo que hace la instrucción `while` la primera vez que se evalúa la expresión? La respuesta, por supuesto, es asignar valores a cada variable en la expresión probada antes que se encuentre la instrucción `while`. Por ejemplo, la siguiente secuencia de instrucciones es válida:

```
cuenta = 1;
while (cuenta <= 10)
  cout << cuenta;
```

Utilizando esta secuencia de instrucciones, se asegura que `cuenta` comience con un valor de 1. Podría asignarse cualquier valor a `cuenta` en la instrucción de asignación, lo importante es asignar algún valor. En la práctica, el valor asignado depende de la aplicación.

Todavía debemos cambiar el valor de `cuenta` de modo que al final podamos salir de la instrucción `while`. Hacer esto requiere una expresión como `cuenta = cuenta + 1` para incrementar el valor de `cuenta` cada vez que se ejecute la instrucción `while`. El hecho que una instrucción `while` proporcione la repetición de una instrucción única no impide que se incluya una instrucción adicional para cambiar el valor de `cuenta`. Todo lo que tiene que hacerse es reemplazar la instrucción única con una instrucción compuesta. Por ejemplo,

```
cuenta = 1;          // inicializa cuenta
while (cuenta <= 10)
{
  cout << cuenta;
  cuenta++;          // incrementa cuenta
}
```

Nótese que, por claridad, se ha colocado cada instrucción en la instrucción compuesta en una línea diferente. Esto es consistente con la convención adoptada para instrucciones compuestas en el capítulo anterior. Ahora se analizará la secuencia anterior de instrucciones.

La primera instrucción de asignación establece `cuenta` igual a 1. Entonces se introduce la instrucción `while` y la expresión se evalúa por primera vez. En vista que el valor de `cuenta` es menor que o igual a 10, la expresión es verdadera y se ejecuta la instrucción compuesta. La primera instrucción en la instrucción compuesta utiliza el objeto `cout` para desplegar el valor de `cuenta`. La siguiente instrucción agrega 1 al valor almacenado en la actualidad

en `cuenta`, haciendo este valor igual a 2. Ahora la instrucción `while` regresa para volver a probar la expresión. En vista que `cuenta` todavía es menor que o igual a 10, la instrucción compuesta se ejecuta de nuevo. Este proceso continúa hasta que el valor de `cuenta` llega a 11. El programa 5.1 ilustra estas instrucciones en un programa real.

Programa 5.1

```
#include <iostream>
using namespace std;

int main()
{
  int cuenta;

  cuenta = 1;              // inicializa cuenta
  while (cuenta <= 10)
  {
    cout << cuenta << "  ";
    cuenta++;              // incrementa cuenta
  }

  return 0;
}
```

La salida del programa 5.1 es:

```
1  2  3  4  5  6  7  8  9  10
```

No hay nada especial respecto al nombre `cuenta` usado en el programa 5.1. Podría haberse utilizado cualquier variable entera válida.

Antes que consideremos otros ejemplos de la instrucción `while`, se ameritan dos comentarios concernientes al programa 5.1. Primero, la instrucción `cuenta++` puede reemplazarse con cualquier instrucción que cambie el valor de `cuenta`. Una instrucción como `cuenta = cuenta + 2`, por ejemplo, causaría que se desplegara cada segundo entero. En segundo lugar, es responsabilidad del programador asegurar que `cuenta` sea cambiada en una forma que conduzca al final a una salida normal de `while`. Por ejemplo, si se reemplaza la expresión `cuenta++` con la expresión `cuenta--`, el valor de cuenta nunca excederá 10 y se creará un ciclo infinito. Un ***ciclo infinito*** es un ciclo que nunca termina; el programa sigue desplegando números hasta que usted se percata que el programa no está funcionando como esperaba.

Ahora que tiene alguna familiaridad con la instrucción `while`, vea si puede leer y determinar la salida del programa 5.2.

Programa 5.2

```
#include <iostream>
using namespace std;

int main()
{
  int i;

  i = 10;
  while (i >= 1)
  {
    cout << i << "  ";
    i--;              // resta 1 de i
  }

  return 0;
}
```

La instrucción de asignación en el programa 5.2 establece inicialmente la variable `int i` en 10. La instrucción `while` verifica entonces si el valor de `i` es mayor que o igual a 1. Mientras la expresión es verdadera, el valor de `i` es desplegado por el objeto `cout` y el valor de `i` es disminuido en 1. Cuando al fin i llega a cero, la expresión es falsa y el programa sale de la instrucción `while`. Por tanto, se obtiene el siguiente despliegue cuando se ejecuta el programa 5.2:

```
10  9  8  7  6  5  4  3  2  1
```

Para ilustrar el poder de la instrucción `while`, considere la tarea de imprimir una tabla de números de 1 a 10 con sus cuadrados y cubos. Esto puede hacerse con una instrucción `while` simple como se ilustra en el programa 5.3.

Cuando se ejecuta el programa 5.3, se produce el siguiente despliegue:

```
NUMERO    CUADRADO    CUBO
------    --------    ----
  1           1          1
  2           4          8
  3           9         27
  4          16         64
  5          25        125
  6          36        216
  7          49        343
  8          64        512
  9          81        729
 10         100       1000
```

Programa 5.3

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  int num;

  cout << "NUMERO    CUADRADO    CUBO\n"
       << "------    --------    ----\n";

  num = 1;
  while (num < 11)
  {
    cout << setw(3) << num << "         "
         << setw(3) << num * num << "       "
         << setw(4) << num * num * num << endl;
    num++;          // incrementa num
  }
  return 0;
}
```

Hay que observar que la expresión usada en el programa 5.3 es `num < 11`. Para la variable entera `num`, esta expresión es exactamente equivalente a la expresión `num <= 10`. La elección de cuál usar es suya por completo.

Si se desea usar el programa 5.3 para producir una tabla de 1000 números, todo lo que se hace es cambiar la expresión en la instrucción `while` de `num < 11` a `num < 1001`. Cambiar el 11 a 1001 produce una tabla de 1000 líneas, nada mal para una simple instrucción `while` de cinco líneas.

Todos los ejemplos de programa que ilustran la instrucción `while` representan ciclos de cuenta fija porque la condición probada es un contador que verifica un número fijo de repeticiones. Puede hacerse una variación en el ciclo de cuenta fija cuando el contador no se incremente en uno cada vez que pasa por el ciclo, sino por algún otro valor. Por ejemplo, considere la tarea de producir una tabla de conversión de temperatura Celsius a Fahrenheit. Suponga que se van a desplegar temperaturas Fahrenheit correspondientes a las temperatu-

ras Celsius que varían de 5 a 50 grados en incrementos de cinco grados. El despliegue deseado puede obtenerse con la serie de instrucciones

```
celsius = 5;     // valor Celsius inicial
while (celsius <= 50)
{
  fahren = (9.0/5.0) * celsius + 32.0;
  cout << celsius << "      "
       << fahren;
  celsius = celsius + 5;
}
```

Como antes, la instrucción `while` consiste en todo desde la palabra `while` hasta la llave de cierre de la instrucción compuesta. Antes de introducir el ciclo `while` nos hemos asegurado de asignar un valor al contador que se va a evaluar, y hay una instrucción para alterar el valor del contador dentro del ciclo (en incrementos de 5 grados Celsius) para asegurar una salida del ciclo `while`. El programa 5.4 ilustra el uso de código similar en un programa completo.

Programa 5.4

```
#include <iostream>
#include <iomanip>
using namespace std;

// un programa para convertir Celsius en Fahrenheit
int main()
{
  const int MAX_CELSIUS = 50;
  const int VAL_INICIAL = 5;
  const int TAMAÑO_PASO = 5;
  int celsius;
  double fahren;

  cout << "GRADOS     GRADOS\n"
       << "CELSIUS  FAHRENHEIT\n"
       << "-------  ----------\n";

  celsius = VAL_INICIAL;

    // establecer los formatos de salida solo para numeros en punto flotante
  cout << setiosflags(ios::showpoint)
       << setprecision(2);
```

(Continúa)

(Continuación)

```
   while (celsius <= MAX_CELSIUS)
   {
     fahren = (9.0/5.0) * celsius + 32.0;
     cout << setw(4)  << celsius << fixed
          << setw(13) << fahren << endl;
     celsius = celsius + TAMAÑO_PASO;
   }

   return 0;
}
```

El despliegue obtenido cuando se ejecuta el programa 5.4 es

```
GRADOS    GRADOS
CELSIUS   FAHRENHEIT
-------   ----------
   5         41.00
  10         50.00
  15         59.00
  20         68.00
  25         77.00
  30         86.00
  35         95.00
  40        104.00
  45        113.00
  50        122.00
```

EJercicios 5.2

1. Vuelva a escribir el programa 5.1 para imprimir los números 2 a 10 en incrementos de dos. La salida de su programa deberá ser

   ```
   2  4  6  8  10
   ```

2. Vuelva a escribir el programa 5.4 para producir una tabla que empiece en un valor Celsius de −10 y termine con un valor Celsius de 60, en incrementos de diez grados.

3. *a.* Para el siguiente programa determine el número total de elementos desplegados. Determine además el primer y último números impresos.

```
#include <iostream>
using namespace std;

int main()
{
  int num = 0;
  while (num <= 20)
  {
    num++;
    cout << num << " ";
  }

  return 0;
}
```

b. Introduzca y ejecute el programa del ejercicio 3a en una computadora para verificar sus respuestas al ejercicio.

c. ¿Cómo afectaría a la salida si las dos instrucciones dentro de la instrucción compuesta se invirtieran (es decir, si la llamada a `cout` se hiciera antes de la instrucción `++ num`)?

4. Escriba un programa en C++ que convierta galones en litros. El programa deberá desplegar galones de 10 a 20 en incrementos de un galón y los litros equivalentes correspondientes. Use la relación que 1 galón contiene 3.785 litros.

5. Escriba un programa en C++ que convierta pies en metros. El programa deberá desplegar pies de 3 a 30 en incrementos de tres pies y los metros equivalentes correspondientes. Use la relación que hay 3.28 pies en un metro.

6. Una máquina comprada por 28 000 dólares se deprecia 4000 dólares por año durante siete años. Escriba y ejecute un programa en C++ que calcule y despliegue una tabla de depreciación para siete años. La tabla deberá tener la forma

Año	Depreciación	Valor al final del año	Depreciación acumulada
1	4000	24000	4000
2	4000	20000	8000
3	4000	16000	12000
4	4000	12000	16000
5	4000	8000	20000
6	4000	4000	24000
7	4000	0	28000

7. Un automóvil viaja a una velocidad promedio de 55 millas por hora durante cuatro horas. Escriba un programa en C++ que despliegue la distancia, en millas, que el automóvil ha recorrido después de 1, 2, etc., horas hasta el final del viaje.

8. a. Una fórmula de conversión aproximada para convertir temperatura Fahrenheit en Celsius es

Celsius = (Fahrenheit – 30) / 2

Usando esta fórmula, y empezando con una temperatura Fahrenheit de cero grados, escriba un programa C++ que determine cuándo la temperatura Celsius equivalente aproximada difiere del valor equivalente exacto por más de cuatro grados. (*Sugerencia:* Use un ciclo `while` que termine cuando la diferencia entre los equivalentes Celsius aproximados y exactos exceda de cuatro grados.)

b. Usando la fórmula de conversión a Celsius aproximada dada en el ejercicio 8a, escriba un programa en C++ que produzca una tabla de temperaturas Fahrenheit, temperaturas Celsius equivalentes exactas y la diferencia entre los valores Celsius equivalentes correctos y aproximados. La tabla deberá comenzar en cero grados Fahrenheit, usar incrementos de dos grados Fahrenheit y terminar cuando la diferencia entre los valores exactos y aproximados difiera por más de cuatro grados.

9. El valor del número de Euler, *e*, puede aproximarse usando la fórmula

$e = 1 + 1/1! + ½! + 1/3! + 1/4! + 1/5! + \cdots$

Usando esta fórmula, escriba un programa en C++ que aproxime el valor de *e* usando un ciclo `while` que termine cuando la diferencia entre dos aproximaciones sucesivas difiera por menos que 10 e–9.

10. El valor del seno de *x* puede aproximarse usando la fórmula

$$\text{sen } x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \cdots$$

Utilizando esta fórmula, determine cuántos términos se necesitan para aproximar el valor devuelto por la función `sin()` intrínseca con un error menor que 1 e–6, cuando $x = 30$ grados. (*Sugerencias:* Use un ciclo `while` que termine cuando la diferencia entre el valor devuelto por la función `sin()` intrínseca y la aproximación es menor que 1 e–6. Además observe que *x* debe convertirse primero a una medida en radianes y que el signo alternante en la serie aproximada puede determinarse como $(-1)^*(n + 1)$ donde *n* es el número de términos usados en la aproximación.)

5.3 CICLOS while INTERACTIVOS

La combinación de la introducción de datos interactivos con las capacidades de repetición de la instrucción `while` produce programas muy adaptables y potentes. Para entender el concepto implicado, considere el programa 5.5, donde una instrucción `while` se usa para aceptar y luego desplegar cuatro números introducidos por el usuario, uno a la vez. Aunque usa una idea muy simple, el programa resalta los conceptos del flujo de control necesarios para producir programas más útiles.

Programa 5.5

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  const int MAXNUMS = 4;
  int cuenta;
  double num;

  cout << "\nEste programa le pedira que introduzca "
       << MAXNUMS << " numeros.\n";
  cuenta = 1;
  while (cuenta <= MAXNUMS)
  {
    cout << "\nIntroduzca un numero: ";
    cin  >> num;
    cout << "El número introducido es " << num;
    count++;
  }
  cout << endl;

  return 0;
}
```

La siguiente es una muestra de la ejecución del programa 5.5.

```
Este programa le pedira que introduzca 4 numeros.

Introduzca un numero: 26.2
El número introducido es 26.2
Introduzca un numero: 5
El número introducido es 5
Introduzca un numero: 103.456
El número introducido es 103.456
Introduzca un numero: 1267.89
El número introducido es 1267.89
```

Revisemos el programa para entender con claridad cómo se produjo la salida. El primer mensaje desplegado es causado por la ejecución del primer objeto `cout` invocado. Esta llamada está afuera de la instrucción `while` y es anterior a ella, así que se ejecuta una vez antes que cualquier instrucción en el ciclo `while`.

Una vez que se introduce el ciclo `while`, las instrucciones dentro de la instrucción compuesta son ejecutadas mientras la condición probada sea verdadera. La primera vez que pasa por la instrucción compuesta, se despliega el mensaje `Introduzca un numero:`. Entonces el programa llama a `cin`, que obliga a la computadora a esperar que se introduzca un número en el teclado. Una vez que se ha introducido un número y se ha oprimido la tecla Retorno o Entrar, el objeto `cout` despliega el número. Entonces la variable `cuenta` se incrementa en uno. Este proceso se repite hasta que se han realizado cuatro pasos a través del ciclo y el valor de `cuenta` es 5. Cada paso causa que se despliegue el mensaje `Introduzca un numero:` causa que se haga una llamada a `cin` y causa que se despliegue el mensaje `El numero introducido` es. La figura 5.4 ilustra este flujo de control.

En lugar de tan sólo desplegar los números introducidos, el programa 5.5 puede modificarse para usar los datos introducidos. Por ejemplo, hagamos que se sumen los números introducidos y se despliegue el total. Para hacer esto, se debe tener mucho cuidado con la forma en que se suman los números, en vista que se usa la misma variable, `num`, para cada número introducido. Debido a esto la introducción de un número nuevo en el programa 5.5 causa de manera automática que se pierda el número previo almacenado en `num`. Por tanto, cada número introducido debe sumarse al total antes que se introduzca otro número. La secuencia requerida es

Introducir un número
Sumar el número al total

¿Cómo se suma un número individual a un total? Una instrucción como `total = total + num` hace el trabajo a la perfección. Ésta es la instrucción de acumulación introducida en la sección 3.1. Después de introducir cada número, la instrucción de acumulación suma el número en el total, como se ilustra en la figura 5.5. El flujo de control completo requerido para sumar los números se ilustra en la figura 5.6.

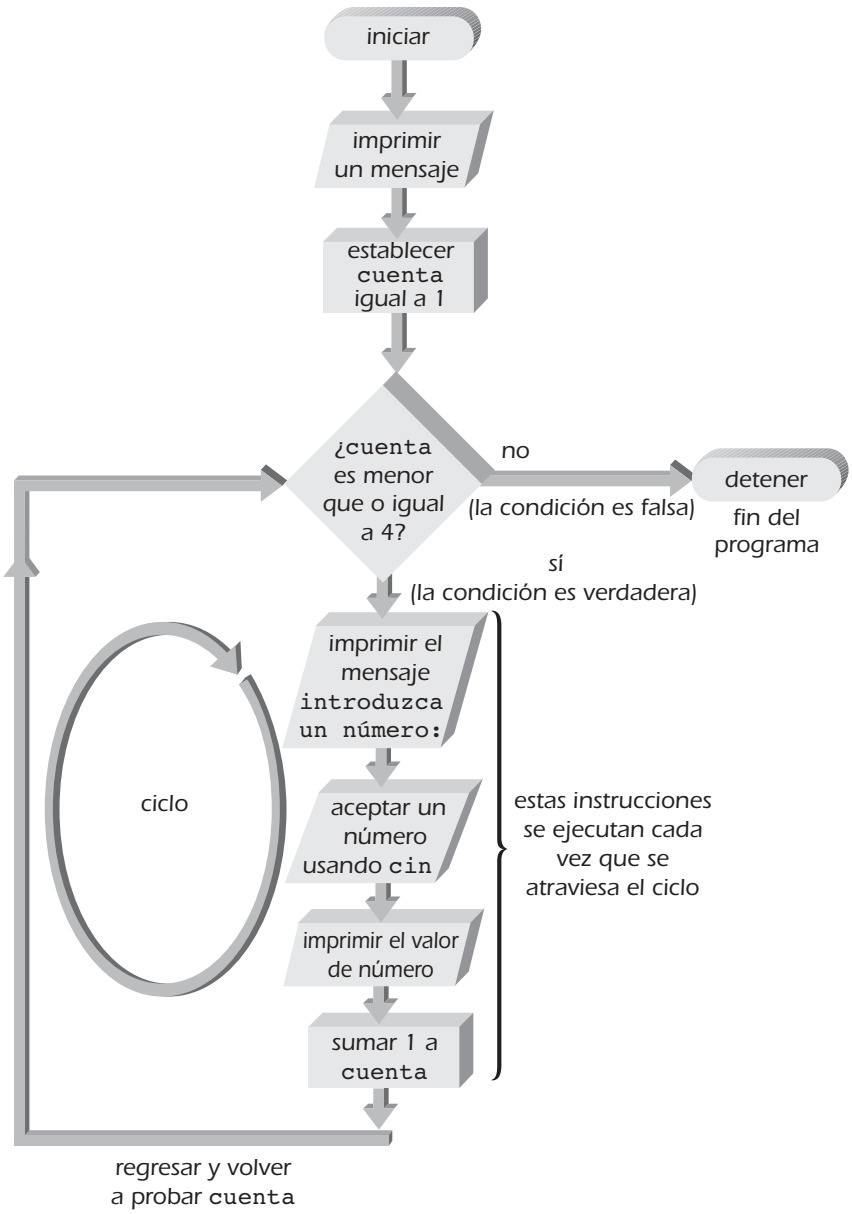


Figura 5.4 *Diagrama de flujo de control para el programa 5.5.*

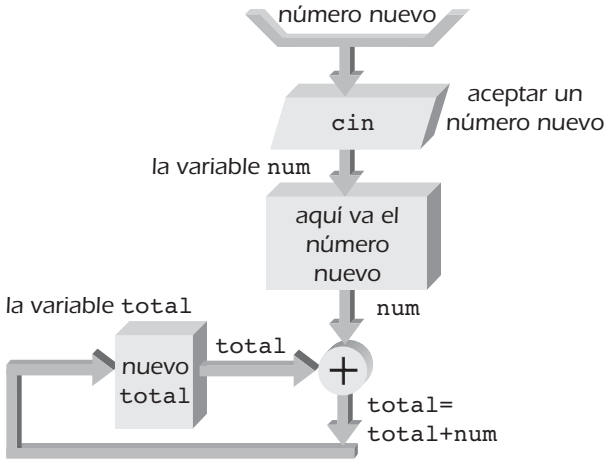


Figura 5.5 *Aceptar y sumar un número a un total.*

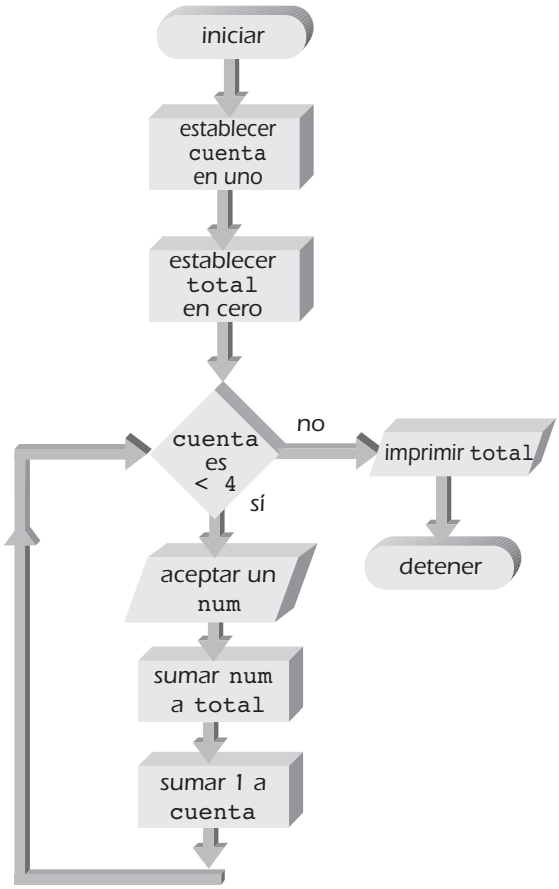


Figura 5.6 *Flujo de control de acumulación.*

Al revisar la figura 5.6, observe que se ha previsto establecer al inicio el total en cero antes que se introduzca el ciclo `while`. Si se quiere despejar el total dentro del ciclo `while`, se establecería en cero cada vez que se ejecutara el ciclo y se borraría cualquier valor almacenado con anterioridad.

El programa 5.6 incorpora las modificaciones necesarias al programa 5.5 para obtener el total de los números introducidos. Como se indica en el diagrama de flujo mostrado en la figura 5.6, la instrucción `total = total + num;` se coloca inmediatamente después de la instrucción `cin`. Poner la instrucción de acumulación en este punto en el programa asegura que el número introducido es "capturado" de inmediato en el total.

Programa 5.6

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  const int MAXNUMS = 4;
  int cuenta;
  double num, total;

  cout << "\nEste programa le pedira que introduzca "
       << MAXNUMS << " numeros.\n";
  cuenta = 1;
  total = 0;

  while (cuenta <= MAXNUMS)
  {
    cout << "\nIntroduzca un numero: ";
    cin  >> num;
    total = total + num;
    cout << "El total es ahora " << setprecision(7) << total;
    cuenta++;
  }

  cout << "\nEl total final es " << setprecision(7) << total << endl;

  return 0;
}
```

Revisemos el programa 5.6. La variable `total` fue creada para almacenar el total de los números introducidos. Antes de introducir la instrucción `while` el valor de `total` se establece en cero. Esto asegura que cualquier valor previo presente en la ubicación o ubicaciones de almacenamiento asignadas a la variable `total` es borrado. Dentro del ciclo `while` la instrucción `total = total + num;` se usa para sumar el valor del número introducido a total. Conforme se introduce cada valor, se suma al `total` existente para crear un total nuevo. Por tanto, `total` se vuelve un subtotal corriente de todos los valores introducidos. Sólo cuando todos los números son introducidos `total` contiene la suma final de todos los números. Después que termina el ciclo `while`, se usa una instrucción `cout` para desplegar esta suma.

Usando los mismos datos que se introdujeron en la muestra de ejecución para el programa 5.5, se hizo la siguiente muestra de ejecución del programa 5.6:

```
Este programa le pedira que introduzca 4 numeros.

Introduzca un numero: 26.2
El número introducido es 26.2
Introduzca un numero: 5
El total es ahora 31.2
Introduzca un numero: 103.456
El total es ahora 134.656
Introduzca un numero: 1267.89
El total es ahora 1402.546

El total final es 1402.546
```

Habiendo usado una instrucción de asignación de acumulación para sumar los números introducidos, ahora se puede ir más allá y calcular el promedio de los números. ¿Dónde se calculará el promedio, dentro del ciclo `while` o fuera de él?

En el presente caso, calcular un promedio requiere que estén disponibles tanto una suma final como el número de elementos en esa suma. Entonces se calcula el promedio dividiendo la suma final entre el número de elementos. En este punto, se debe preguntar: "¿En qué punto en el programa está disponible la suma correcta y en qué punto está disponible el número de elementos?" Al revisar el programa 5.6 se puede observar que la suma correcta necesaria para calcular el promedio está disponible después que termina el ciclo `while`. De hecho, el propósito del ciclo `while` es asegurar que los números son introducidos y sumados en forma correcta para producir una suma correcta. Después que termina el ciclo, también tenemos una cuenta del número de elementos usados en la suma. Sin embargo, debido a la forma en que fue construido el ciclo `while`, el número en `cuenta` (5) cuando termina el ciclo es 1 más que el número de elementos (4) usados para obtener el total. Sabiendo esto, tan sólo se resta uno de cuenta antes de utilizarlo para determinar el promedio. Con esto como antecedentes, veamos si puede leer y entender el programa 5.7.

Programa 5.7

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  const int MAXNUMS = 4;
  int cuenta;
  double num, total, promedio;

  cout << "\nEste programa le pedira que introduzca "
       << MAXNUMS << " numeros.\n";
  cuenta = 1;
  total = 0;

  while (cuenta <= MAXNUMS)
  {
    cout << "Introduzca un número: ";
    cin  >> num;
    total = total + num;
    cuenta++;
  }

  cuenta--;
  promedio = total / cuenta;
  cout << "\nEl promedio de los numeros es " << promedio << endl;

  return 0;
}
```

El programa 5.7 es casi idéntico al programa 5.6, excepto por el cálculo del promedio. También se ha eliminado el despliegue constante del total dentro y después del ciclo `while`. El ciclo en el programa 5.7 se usa para introducir y sumar cuatro números. Inmediatamente después de salir del ciclo, se calcula y se despliega el promedio.

A continuación se presenta una muestra de ejecución del programa 5.7:

```
Este programa le pedira que introduzca 4 numeros.
Introduzca un numero: 26.2
Introduzca un numero: 5
Introduzca un numero: 103.456
Introduzca un numero: 1267.89

El promedio de los numeros es 350.637
```

Centinelas

Todos los ciclos que hemos creado hasta ahora han sido ejemplos de ciclos de cuenta fija, donde se ha usado un contador para controlar el número de iteraciones del ciclo. Por medio de una instrucción `while`, también pueden construirse ciclos de condición variable. Por ejemplo, cuando se introducen calificaciones puede ser que no se desee contar el número de calificaciones que se introducirán, sino que se preferiría introducirlas en forma continua y, al final, introducir un valor de datos especial para señalar el final de la introducción de datos.

En la programación por computadora, los valores de datos usados para señalar el inicio o el fin de una serie de datos se llaman **centinelas**. Los valores centinelas deben seleccionarse, por supuesto, de modo que no entren en conflicto con los valores de datos legítimos. Por ejemplo, si se fuera a elaborar un programa para procesar las calificaciones de un estudiante, y suponiendo que no se dan créditos extras que pudieran producir una calificación mayor que 100, podría usarse cualquier calificación mayor que 100 como un valor centinela. El programa 5.8 ilustra este concepto. En el programa 5.8 se solicitan y aceptan datos en forma continua hasta que se introduce un número mayor que 100. La introducción de un número mayor que 100 alerta al programa para que salga del ciclo `while` y despliegue la suma de los números introducidos.

A continuación puede verse una muestra de ejecución usando el programa 5.8. En tanto se introduzcan calificaciones menores que o iguales a 100, el programa continúa solicitando y aceptando datos adicionales. Cuando se introduce un número menor que o igual a 100, el programa suma este número al total. Cuando se introduce un número mayor que 100, se sale del ciclo y se despliega la suma de las calificaciones que se introdujeron.

```
Para dejar de introducir calificaciones, introduzca
 cualquier numero mayor que 100.

Introduzca una calificacion: 95
Introduzca una calificacion: 100
Introduzca una calificacion: 82
Introduzca una calificacion: 101

El total de las calificaciones es 277
```

Programa 5.8

```
#include <iostream>
using namespace std;

int main()
{
  const int CALIF_ALTA = 100;
  double calificacion, total;

  calificacion = 0;
  total = 0;
  cout << "\nPara dejar de introducir calificaciones, introduzca cualquier numero";
  cout << "\n mayor que 100.\n\n";

  while (calificacion <= CALIF_ALTA)
  {
    total = total + calificacion;
    cout << "Introduzca una calificacion: ";
    cin  >> calificacion;
  }

  cout << "\nEl total de las calificaciones es " << total << endl;
  return 0;
}
```

Instrucciones `break` y `continue`

Dos instrucciones útiles en conexión con las instrucciones de repetición son las instrucciones **`break`** y **`continue`**. Se ha encontrado la instrucción `break` en relación con la instrucción `switch`. La sintaxis de esta instrucción es

break;

Una instrucción `break`, como su nombre implica, obliga a una interrupción inmediata, o salida, de `switch`, `while` y las instrucciones `for` y `do-while` presentadas en las siguientes secciones.

Por ejemplo, la ejecución del siguiente ciclo `while` es terminada de inmediato si se introduce un número mayor que 76.

```
while(cuenta <= 10)
{
  cout << "Introduzca un número: ";
  cin  >> num;
  if (num > 76)
  {
    cout << "¡Perdiste!\n";
    break;       // interrumpe el ciclo
  }
  else
    cout << "¡Sigue intentandolo!\n";
  cuenta++
}
// break salta hasta aquí
```

La instrucción `break` viola los principios puros de la programación estructurada porque proporciona una segunda salida no estándar de un ciclo. No obstante, la instrucción `break` es muy útil y valiosa para interrumpir ciclos cuando se detecta una condición inusual. La instrucción `break` también se usa para salir de una instrucción `switch`, pero esto se debe a que el caso deseado se ha detectado y procesado.

La instrucción `continue` es similar a la instrucción `break` pero sólo se aplica a ciclos creados con instrucciones `while`, `do-while` y `for`. El formato general de una instrucción `continue` es

continue;

Cuando `continue` se encuentra en un `ciclo`, la siguiente iteración del ciclo comienza de inmediato. Para ciclos `while` esto significa que la ejecución es transferida de manera automática al principio del ciclo y se inicia una reevaluación de la expresión probada. Aunque la instrucción `continue` no tiene un efecto directo en una instrucción `switch`, puede incluirse dentro de una instrucción `switch` que esté contenida en un `ciclo`. Aquí el efecto de `continue` es el mismo: se comienza la siguiente iteración del ciclo.

Como una regla general, la instrucción `continue` es menos útil que la instrucción `break`, pero es conveniente para saltarse datos que no deberían ser procesados mientras permanecen en un ciclo. Por ejemplo, las calificaciones inválidas simplemente son ignoradas en la siguiente sección de código y sólo se suman las calificaciones válidas al total:[2]

[2]Sin embargo, la instrucción `continue` no es esencial y la selección podría haberse escrito como

```
if (calificacion >= 0 && calificacion <= 100)
{
  total = total + calificacion;
  cuenta++;
}
```

```
while (cuenta < 30)
{
  cout << "Introduzca una calificacion: ";
  cin  >> calificacion;
  if(calificacion < 0 || calificacion > 100)
    continue;
  total = total + calificacion;
  cuenta++;
}
```

La instrucción nula

Todas las instrucciones deben terminarse con un punto y coma. Un punto y coma sin nada que lo preceda también es una instrucción válida, llamada instrucción nula. Por tanto, la instrucción

```
;
```

es una instrucción nula. Ésta es una instrucción que no hace nada y que se utiliza donde se requiere una instrucción desde el punto de vista sintáctico, pero no invoca ninguna acción. Las instrucciones nulas se usan de manera típica con instrucciones `while` o `for`. Un ejemplo de una instrucción `for` que usa una instrucción nula se encuentra en el programa 5.10c en la siguiente sección.

Ejercicios 5.3

1. Vuelva a escribir el programa 5.6 para calcular el total de ocho números.
2. Vuelva a escribir el programa 5.6 para desplegar el indicador:

   ```
   Por favor introduzca el número total de valores de datos
   que se van a sumar:
   ```

   En respuesta a este indicador, el programa deberá aceptar un número introducido por un usuario y luego usar este número para controlar el número de veces que se ejecuta el ciclo `while`. Por tanto, si el usuario introduce *5* en respuesta al indicador, el programa deberá solicitar la introducción de cinco números y desplegar el total después que se hayan introducido los cinco números.
3. **a.** Escriba un programa en C++ para convertir grados Celsius a Fahrenheit. El programa deberá solicitar el valor Celsius inicial, el número de conversiones que se harán y el incremento entre valores Celsius. El despliegue deberá tener encabezados apropiados y enlistar el valor Celsius y el valor Fahrenheit correspondiente. Use la relación

   $$Fahrenheit = (9.0 / 5.0) * Celsius + 32.0$$

   b. Ejecute el programa escrito en el ejercicio 3a en una computadora. Verifique que su programa empiece en el valor Celsius inicial correcto y contenga el número exacto de conversiones especificado en sus datos de entrada.
4. **a.** Modifique el programa escrito en el ejercicio 3a para solicitar el valor Celsius inicial, el valor Celsius final y el incremento. Por tanto, en lugar que la condición verifique una cuenta fija, verificará el valor Celsius final.

b. Ejecute el programa escrito en el ejercicio 4a en una computadora. Verifique que su salida comience en el valor inicial correcto y termine en el valor final correcto.

5. Vuelva a escribir el programa 5.7 para calcular el promedio de diez números.

6. Vuelva a escribir el programa 5.7 para desplegar el siguiente indicador:

```
Por favor introduzca el número total de valores de datos
que se van a promediar:
```

En respuesta a este indicador, el programa deberá aceptar un número introducido por un usuario y luego usar este número para controlar el número de veces que se ejecutará el ciclo `while`. Por tanto, si el usuario introduce 6 en respuesta al indicador, el programa deberá solicitar la introducción de seis números y desplegar el promedio de los siguientes seis números introducidos.

7. Por error, un programador puso la instrucción `promedio = total / cuenta;` dentro del ciclo `while` inmediatamente después de la instrucción `total = total + num;` en el programa 5.7. Por tanto, el ciclo while se vuelve

```
while (cuenta <= MAXNUMS)
{
  cout << "Introduzca un numero: ";
  cin  >> num;
  total = total + num;
  promedio = total / cuenta;
  cuenta++;
}
```

¿El programa producirá el resultado correcto con este ciclo `while`?
Desde una perspectiva de programación, ¿cuál ciclo `while` es mejor usar y por qué?

8. Una serie aritmética se define por

$$a + (a + d) + (a + 2d) + (a + 3d) + \cdots + [(a + (n - 1)d)]$$

donde a es el primer término, d es la "diferencia común" y n es el número de términos que se van a sumar. Usando esta información, escriba un programa en C++ que use un ciclo `while` para desplegar cada término y para determinar la suma de la serie aritmética si se tiene que $a = 1$, $d = 3$ y $n = 100$. Asegúrese que su programa despliegue el valor que ha calculado.

9. Una serie geométrica se define por

$$a + ar + ar^2 + ar^3 + \cdots + ar^{n-1}$$

donde a es el primer término, r es la "razón común" y n es el número de términos en la serie. Usando esta información, escriba un programa en C++ que utilice un ciclo `while` para desplegar cada término y para determinar la suma de una serie geométrica si se tiene que $a = 1$, $r = .5$ y $n = 10$. Asegúrese que su programa despliega el valor que se ha calculado.

10. Además del promedio aritmético de un conjunto de números, se puede calcular una media geométrica y una media armónica. La media geométrica de un conjunto de n números $x_1, x_2, \ldots x_n$ se define como

$$\sqrt[n]{x_1 \cdot x_2 \cdot \cdots \cdot x_n}$$

y la media armónica como

$$\frac{n}{\frac{1}{x_1}+\frac{1}{x_2}+\cdots+\frac{1}{x_n}}$$

Usando estas fórmulas, escriba un programa en C++ que continúe aceptando números hasta que se introduzca el número 999 y luego calcule y despliegue tanto la media geométrica como la armónica de los números introducidos. (*Sugerencia:* Será necesario que su programa cuente en forma correcta el número de valores introducidos.)

11. a. Los siguientes datos se recolectaron en un viaje reciente en automóvil.

Millaje	Gallones
22495	Full tank
22841	12.2
23185	11.3
23400	10.5
23772	11.0
24055	12.2
24434	14.7
24804	14.3
25276	15.2

Escriba un programa en C++ que acepte un valor de millaje y galones y calcule las millas por galón (mpg) logradas para ese segmento del viaje. Las millas por galón se obtienen como la diferencia en millaje entre llenadas del tanque dividido entre el número de galones de gasolina utilizados desde que se llenó el tanque.

b. Modifique el programa escrito para el ejercicio 11a para calcular y desplegar adicionalmente las mpg acumuladas después de cada llenada de tanque. Las mpg acumuladas se calculan como la diferencia entre el millaje en cada llenada y el millaje al principio del viaje dividido entre la suma de los galones usados hasta ese punto en el viaje.

5.4 Ciclos `for`

En C++, un **`ciclo for`** se construye usando una instrucción `for`. Esta instrucción realiza las mismas funciones que la instrucción `while`, pero usa una forma diferente. En muchas situaciones, en especial aquellas que usan una condición de cuenta fija, el formato de la instrucción `for` es más fácil de usar que su instrucción `while` equivalente.

La sintaxis de la instrucción `for` es

```
for (lista de inicialización; expresión; lista de alteración)
    instrucción;
```

Aunque la instrucción `for` parece un poco complicada, en realidad es bastante simple si se considera cada una de sus partes por separado.

Dentro del paréntesis de la instrucción `for` hay tres elementos, separados por puntos y comas. Cada uno de estos elementos es opcional y puede describirse de manera individual, pero los puntos y comas deben estar presentes siempre.

En su forma más común, la lista de *inicialización* consiste de una sola instrucción usada para establecer el comienzo (valor inicial) de un contador, la *expresión* contiene el valor máximo o mínimo que puede tener el contador y determina cuándo se termina el ciclo, y la lista de *alteración* proporciona el valor de incremento que se suma o se resta del contador cada vez que se ejecuta el ciclo. Son ejemplos de instrucciones `for` simples que tienen esta forma

```
for (cuenta = 1; cuenta < 10; cuenta = cuenta + 1)
  cout << cuenta;
```

y

```
for (i = 5; i <= 15; i = i + 2)
  cout << i;
```

En la primera instrucción `for`, la variable contadora se llama `cuenta`, el valor inicial asignado a `cuenta` es 1, el ciclo continúa en tanto el valor en `cuenta` sea menor que 10, y el valor de `cuenta` se incrementa en uno cada vez que pasa por el ciclo. En la siguiente instrucción `for`, la variable contadora es nombrada `i`, el valor inicial asignado a `i` es 5, el ciclo continúa en tanto el valor de i sea menor que o igual a 15 y el valor de `i` es incrementado en 2 cada vez que pasa por el ciclo. En ambos casos se usa una instrucción `cout` para desplegar el valor del contador. Otro ejemplo de un ciclo `for` está dado en el programa 5.9.

Cuando se ejecuta el programa 5.9, se produce el siguiente despliegue:

```
NUMERO     RAIZ CUADRADA
------     -------------
   1          1.00000
   2          1.41421
   3          1.73205
   4          2.00000
   5          2.23607
```

Las primeras dos líneas desplegadas por el programa son producidas por las dos instrucciones `cout` colocadas antes de la instrucción `for`. La salida restante es producida por el ciclo `for`. Este ciclo comienza con la instrucción `for` y es ejecutado como sigue.

El valor inicial asignado a la variable contadora `cuenta` es 1. En vista que el valor en cuenta no excede del valor final de 5, la ejecución de la instrucción `cout` dentro del ciclo produce el despliegue

```
1          1.00000
```

Entonces se transfiere el control de nuevo a la instrucción `for`, la cual entonces incrementa el valor en `cuenta` a 2, y el ciclo se repite, produciendo el despliegue

```
2          1.41421
```

Programa 5.9

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  const int CUENTAMAX = 5;
  int cuenta;

  cout << "NUMERO    RAIZ CUADRADA\n";
  cout << "------    -----------\n";

  cout << setiosflags(ios::showpoint);
  for (cuenta = 1; cuenta <= CUENTAMAX; cuenta++)
    cout << setw(4) << cuenta
         << setw(15) << sqrt(double(cuenta)) << endl;

  return 0;
}
```

Este proceso continúa hasta que el valor en cuenta excede el valor final de 5, produciendo la tabla de salida completa. Con propósitos de comparación, un ciclo `while` equivalente al ciclo `for` contenido en el programa 5.9 es:

```
cuenta = 1
while (cuenta <= MAXCOUNT)
{
  cout << setw(4) << cuenta
       << setw(15) << sqrt(cuenta) << endl;
  cuenta++;
}
```

Como se puede observar en este ejemplo, la diferencia entre los ciclos `for` y `while` es la colocación de los elementos de inicialización, prueba de condición e incremento. El agrupamiento de estos elementos en la instrucción `for` es muy conveniente cuando deben construirse ciclos de cuenta fija. Vea si puede determinar la salida producida por el programa 5.10.

Programa 5.10

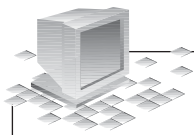
```
#include <iostream>
using namespace std;

int main()
{
  int cuenta;

  for (cuenta = 2; cuenta <= 20; cuenta = cuenta + 2)
    cout << cuenta << "  ";

  return 0;
}
```

¿Pudo encontrar la respuesta? El ciclo comienza con una cuenta inicializada en 2, se detiene cuando cuenta excede de 20 e incrementa cuenta en pasos de 2. La salida del programa 5.10 es

```
2  4  6  8  10  12  14  16  18  20
```

La instrucción `for` no requiere que cualquiera de los elementos entre paréntesis esté presente o que sean usados para inicializar o alterar los valores en las instrucciones de expresión. Sin embargo, los dos puntos y comas deben estar presentes dentro del paréntesis de `for`. Por ejemplo, la construcción `for ( ; cuenta <= 20 ;)` es válida.

Si falta la lista de inicialización, el paso de inicialización se omite cuando se ejecuta la instrucción `for`. Esto significa, por supuesto, que el programador debe proporcionar las inicializaciones requeridas antes que se encuentre la instrucción `for`. Del mismo modo, si falta la lista de alteración, cualesquiera expresiones necesarias para alterar la evaluación de la expresión probada deben incluirse en forma directa dentro de la parte de instrucción del ciclo. La instrucción `for` sólo asegura que todas las expresiones en la lista de inicialización se ejecutan una vez, antes de la evaluación de la expresión probada, y que todas las expresiones en la lista de alteración se ejecuten al final del ciclo antes que se vuelva a verificar la expresión probada. Por tanto, el programa 5.10 puede volverse a escribir en cualquiera de las tres formas mostradas en los programas 5.10a, 5.10b y 5.10c.

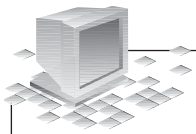


Programa 5.10a

```
#include <iostream>
using namespace std;

int main()
{
  int cuenta;

  cuenta = 2;    // inicializador fuera de la instrucción for
  for ( ; cuenta <= 20; cuenta = cuenta + 2)
    cout << cuenta << "  ";

  return 0;
}
```

Programa 5.10b

```
#include <iostream>
using namespace std;

int main()
{
  int cuenta;

  cuenta = 2;    // inicializador fuera del ciclo for
  for( ; count <= 20; )
  {
    cout << cuenta << "  ";
    cuenta = cuenta + 2;     // instrucción de alteración
  }

  return 0;
}
```



Programa 5.10c

```
#include <iostream>
using namespace std;

int main()   // todas las expresiones dentro de los paréntesis de for
{
  int cuenta;

  for (cuenta = 2; cuenta <= 20; cout << cuenta << "  ", cuenta = cuenta + 2);

  return 0;
}
```

En el programa 5.10a, `cuenta` es inicializada fuera de la instrucción `for` y la primera lista dentro del paréntesis se deja en blanco. En el programa 5.10b, tanto la lista de inicialización como la lista de alteración son eliminadas de dentro del paréntesis. El programa 5.10b también usa una instrucción compuesta dentro del ciclo `for`, con la instrucción de alteración de la expresión incluida en la instrucción compuesta. Por último, el programa 5.10c ha incluido todos los elementos dentro del paréntesis, de modo que no hay necesidad de ninguna instrucción útil después del paréntesis. Aquí la instrucción nula satisface el requerimiento sintáctico de que una instrucción siga al paréntesis de `for`.

Observe también en el programa 5.10c que la lista de alteración (último conjunto de elementos en el paréntesis) consiste en dos elementos, y que se ha usado una coma para separar estos elementos. El uso de comas para separar elementos en las listas de inicialización y de alteración se requiere si cualquiera de estas dos listas contiene más de un elemento. Por último, nótese el hecho que los programas 5.10a, 5.10b y 5.10c son inferiores al programa 5.10, y aunque pueda encontrarlos en su carrera de programación, no debería usarlos. Agregar elementos distintos a las variables de control del ciclo y sus condiciones de actualización dentro de la instrucción `for` tiende a confundir su legibilidad y puede introducir efectos indeseados. Mantener "limpia" la estructura de control del ciclo, como se hizo en el programa 5.10, es importante y es una buena práctica de programación.

Aunque las listas de inicialización y alteración pueden omitirse de una instrucción `for`, omitir la expresión probada produce un ciclo infinito. Por ejemplo, un ciclo así es creado por la instrucción

```
for (cuenta = 2; ; cuenta = cuenta + 1)
  cout << cuenta;
```

Como con la instrucción `while`, pueden usarse las instrucciones `break` y `continue` dentro de un ciclo `for`. Un `break` obliga a una salida inmediata del ciclo `for`, como lo hace en el ciclo `while`. Sin embargo, un `continue` obliga a que el control se pase a la lista de alteración en una instrucción `for`, después de lo cual se reevalúa la expresión probada. Esto difiere de la acción de un `continue` en una instrucción `while`, donde el control se pasa en forma directa a la reevaluación de la expresión probada.

Punto de Información

Dónde colocar las llaves de apertura

Hay dos estilos para escribir ciclos `for` que son usados por programadores profesionales en C++. Estos estilos sólo se utilizan cuando el ciclo `for` contiene una instrucción compuesta. El estilo ilustrado y usado en el texto toma la forma

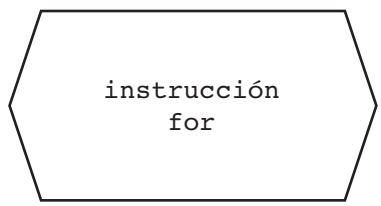
```
for (expresión)
{
   la instrucción compuesta va aquí
}
```

Un estilo igual de aceptable que ha sido usado por muchos programadores coloca la llave inicial de la instrucción compuesta en la primera línea. Usando este estilo, un ciclo `for` aparece como

```
for (expresión) {
   la instrucción compuesta va aquí
}
```

La ventaja del primer estilo es que las llaves se alinean una bajo la otra, facilitando localizar los pares de llaves. La ventaja del segundo estilo es que hace más compacto el código y ahorra una línea, permitiendo que se vea más código en la misma área de despliegue. Ambos estilos se usan pero casi nunca se mezclan. Seleccione cualquier estilo que le atraiga y sea consistente en su uso. Como siempre, la sangría que use dentro de la instrucción compuesta (dos o cuatro espacios, o un tabulador) también deberá ser consistente en todos sus programas. La combinación de estilos que seleccione se convierte en una "firma" para su trabajo de programación.

La figura 5.7 ilustra el funcionamiento interno de un ciclo `for`. Como se muestra, cuando el ciclo `for` es completado, el control se transfiere a la primera instrucción ejecutable que se encuentre después del ciclo. Para evitar ilustrar siempre estos pasos de manera forzada, se dispone de un conjunto simplificado de símbolos de diagrama de flujo para describir los ciclos `for`. Usando el hecho que una instrucción for puede representarse con el símbolo de diagrama de flujo



los ciclos `for` completos pueden ilustrarse en forma alternativa como se muestra en la figura 5.8.

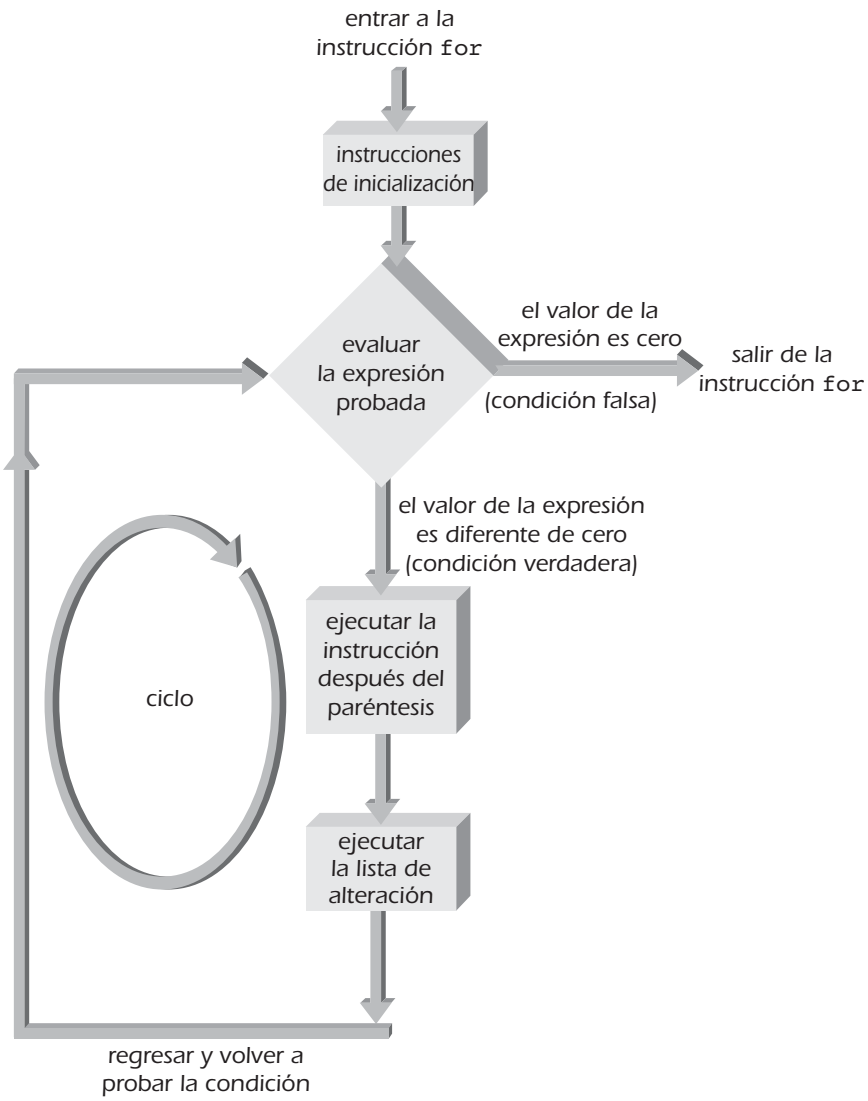


Figura 5.7 *Diagrama de flujo de un ciclo* `for`.

Punto de Información

¿Se debe utilizar un ciclo `for` o uno `while`?

Una pregunta que plantean por lo común los programadores principiantes es cuál estructura de ciclo deberían usar, un ciclo `for` o uno `while`. Ésta es una buena pregunta porque ambas estructuras de ciclo son ciclos de prueba preliminar, que, en C++, pueden usarse para construir tanto ciclos de cuenta fija como ciclos de condición variable.

En casi todos los otros lenguajes de computadora, incluyendo Visual Basic y Pascal, la respuesta es relativamente sencilla, porque la instrucción `for` sólo puede usarse para construir ciclos de cuenta fija. Por tanto, en estos lenguajes las instrucciones `for` se usan para construir ciclos de cuenta fija y las instrucciones `while` se usan por lo general sólo cuando se construyen ciclos de condición variable.

En C++, esta distinción tan fácil no se sostiene, pues cada instrucción puede usarse para crear cada tipo de ciclo. La respuesta en C++, entonces, en realidad es cuestión de estilo. En vista que un ciclo `for` y uno `while` son intercambiables en C++, cualquier ciclo es apropiado. Algunos programadores profesionales siempre usan una instrucción `for` para todos los ciclos de prueba preliminar que crean y casi nunca usan una instrucción `while`; otros siempre utilizan una instrucción `while` y rara vez una instrucción `for`. Un tercer grupo tiende a conservar la convención utilizada en otros lenguajes: un ciclo `for` se usa por lo general para crear ciclos de cuenta fija y un ciclo `while` para crear ciclos de condición variable. En C++ todo es cuestión de estilo y encontrará los tres estilos en su carrera de programación.

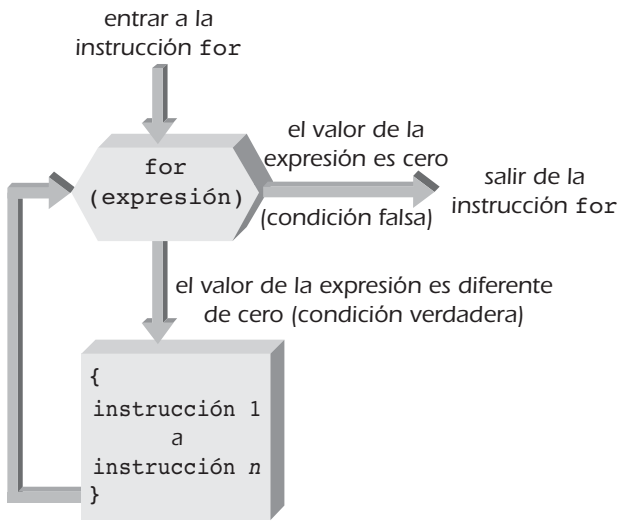


Figura 5.8 *Diagrama de flujo simplificado de un ciclo* `for`.

Para entender el poder enorme de los ciclos `for`, considere la tarea de imprimir una tabla de números del 1 al 10, incluyendo sus cuadrados y cubos, usando esta instrucción. Dicha tabla fue producida con anterioridad usando un ciclo `while` en el programa 5.3. Puede desear revisar el programa 5.3 y compararlo con el programa 5.11 para obtener un sentido mayor de la equivalencia entre los ciclos `for` y `while`.

Programa 5.11

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  const int MAXNUMS = 10;
  int num;

  cout << "NUMERO   CUADRADO   CUBO\n"
       << "------   --------   ----\n"

  for (num = 1; num <= MAXNUMS; num++)
    cout << setw(3) << num << "         "
         << setw(3) << num * num << "       "
         << setw(4) << num * num * num << endl;

  return 0;
}
```

Cuando el programa 5.11 se ejecuta, el despliegue producido es

```
NUMERO     CUADRADO     CUBO
------     --------     ----
  1            1          1
  2            4          8
  3            9         27
  4           16         64
  5           25        125
  6           36        216
  7           49        343
  8           64        512
  9           81        729
 10          100       1000
```

Con sólo cambiar el número 10 en la instrucción `for` del programa 5.11 a 1000 crea un ciclo que se ejecuta 1000 veces y produce una tabla de números del 1 al 1000. Como con la instrucción `while`, este pequeño cambio produce un aumento inmenso en el procesamiento y salida proporcionados por el programa. Nótese también que la expresión `num++` se usó en la lista de alteración en lugar del usual `num = num + 1`.

Ejercicios 5.4

1. Escriba instrucciones `for` individuales para los siguientes casos.
   a. Al usar un contador nombrado `i` que tiene un valor inicial de 1, un valor final de 20 y un incremento de 1.
   b. Al usar un contador nombrado `icuenta` que tiene un valor inicial de 1, un valor final de 20 y un incremento de 2.
   c. Al usar un contador nombrado `j` que tiene un valor inicial de 1, un valor final de 100 y un incremento de 5.
   d. Al usar un contador nombrado `icuenta` que tiene un valor inicial de 20, un valor final de 1 y un incremento de –1.
   e. Al usar un contador nombrado `icuenta` que tiene un valor inicial de 20, un valor final de 1 y un incremento de –2.
   f. Al usar un contador nombrado `cuenta` que tiene un valor inicial de 1.0, un valor final de 16.2 y un incremento de 0.2.
   g. Al usar un contador nombrado `xcnt` que tiene un valor inicial de 20.0, un valor final de 10.0 y un incremento de –0.5.

2. Determine el número de veces que se ejecuta cada ciclo for para la instrucción for escrita para el ejercicio 1.

3. Determine el valor en total después que se ejecuta cada uno de los siguientes ciclos.

   a.
   ```
   total = 0;
   for (i = 1; i <= 10; i = i + 1)
     total = total + 1;
   ```
   b.
   ```
   total = 1;
   for (cuenta = 1; cuenta <= 10; cuenta = cuenta + 1)
     total = total * 2;
   ```
   c.
   ```
   total = 0
   for (i = 10; i <= 15; i = i + 1)
     total = total + i;
   ```
   d.
   ```
   total = 50
   for (i = 1; i <=10; i = i + 1)
     total = total - i;
   ```
   e.
   ```
   total = 1
   for (icnt = 1; icnt <= 8; ++icnt)
     total = total * icnt;
   ```
   f.
   ```
   total = 1.0
   for (j = 1; j <= 5; ++j)
     total = total / 2.0;
   ```

4. Determine la salida del siguiente programa.

```
#include <iostream>
using namespace std;

int main()
{
  int i;

  for (i = 20; i >= 0; i = i - 4)
    cout << i << " ";

  return 0;
}
```

5. Modifique el programa 5.11 para producir una tabla de los números cero a 20 en incrementos de 2, con sus cuadrados y cubos.

6. Modifique el programa 5.11 para producir una tabla de números de 10 a 1, en lugar de 1 a 10 como lo hace ahora.

7. Escriba y ejecute un programa en C++ que despliegue una tabla de 20 conversiones de temperatura de Fahrenheit a Celsius. La tabla deberá comenzar con un valor Fahrenheit de 20 grados e incrementarse en valores de 4 grados. Recuerde que

Celsius = (5.0/9.0) * (*Fahrenheit* – 32)

8. Modifique el programa escrito para el ejercicio 7 para solicitar al inicio el número de conversiones que se van a hacer.

9. La expansión de un puente de acero conforme se calienta hasta una temperatura Celsius final, T_F, desde una temperatura Celsius inicial, T_0, puede aproximarse usando la fórmula

Aumento de longitud = $a * L * (T_F - T_0)$

donde a es el coeficiente de expansión (el cual para el acero es 11.7 e-6) y L es el largo del puente a la temperatura T_0. Usando esta fórmula, escriba un programa en C++ que despliegue una tabla de longitudes de expansión para un puente de acero que tiene 7365 metros de largo a 0 grados Celsius, conforme la temperatura incrementa a 40 grados en incrementos de 5 grados.

10. La probabilidad que una llamada telefónica individual dure menos de t minutos puede aproximarse por la función de probabilidad exponencial

Probabilidad que una llamada dure menos de t minutos = $1 - e^{-t/a}$

donde a es la duración de la llamada promedio y e es el número de Euler (2.71828). Por ejemplo, suponiendo que la duración de la llamada promedio es 2 minutos, la probabilidad que una llamada durará menos de 1 minuto se calcula como $1 - e^{-1/2} = 0.3297$.

Usando esta ecuación de probabilidad, escriba un programa en C++ que calcule y despliegue una lista de probabilidades de la duración de una llamada menor que 1 a menor que 10 minutos, en incrementos de un minuto.

11. a. El índice de llegadas de clientes en un banco concurrido en Nueva York puede estimarse usando la función de probabilidad de Poisson

$$P(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

donde x = el número de clientes que llegan por minuto; λ = el número promedio de llegadas por minuto y e el número de Euler (2.71828). Por ejemplo, si el número promedio de clientes que entran en el banco es de tres clientes por minuto, entonces λ es igual a tres. Por tanto, la probabilidad que un cliente llegue en cualquier minuto es

$$P(x = 1) = \frac{3^1 e^{-3}}{1!} = 0.149561$$

y la probabilidad que lleguen dos clientes en cualquier minuto es

$$P(x = 2) = \frac{3^2 e^{-3}}{2!} = 0.224454$$

Usando la función de probabilidad de Poisson, escriba un programa en C++ que calcule y despliegue la probabilidad de llegada de 1 a 10 clientes en cualquier minuto cuando el índice de llegadas promedio es de tres clientes por minuto.

b. La fórmula dada en el ejercicio 11a también es aplicable para estimar el índice de llegadas de aviones en un aeropuerto concurrido (aquí, un "cliente" que llega es un avión que aterriza). Usando esta misma fórmula, modifique el programa escrito en el ejercicio 11a para aceptar el índice de llegadas promedio como un elemento de datos de entrada. Luego ejecute el programa modificado para determinar la probabilidad de cero a 10 aviones que intenten aterrizar en cualquier periodo de un minuto en un aeropuerto durante las horas de mayor tráfico de llegadas. Suponga que el índice de llegadas promedio para las horas de mayor tráfico es de dos aviones por minuto.

12. Una pelota de golf es soltada desde un avión. La distancia, d, que cae la pelota en t segundos está dada por la ecuación $d = (\frac{1}{2})gt^2$, donde g es la aceleración debida a la gravedad y es igual a 32 pies/s^2. Usando esta información, escriba y ejecute un programa en C++ que despliegue la distancia que cae en cada intervalo de un segundo para diez segundos y la distancia que cae la pelota de golf al final de cada intervalo. La salida deberá completar la siguiente tabla:

Tiempo	Distancia en el intervalo actual	Distancia total
0	0.0	0.0
1	16.0	16.0
.	.	.
.	.	.
.	.	.
10	.	.

13. Suponga que el avión en el ejercicio 12 vuela a una altura de 50 000 pies. Modifique el programa escrito para el ejercicio 12 para determinar cuánto tiempo le tomará a la pelota llegar al suelo. A fin de incrementar la precisión de su resultado sin un número indebido de cálculos, disminuya el intervalo de tiempo de 1 segundo a 0.1 segundos conforme la pelota se acerca al suelo.

14. La secuencia de Fibonacci es 0, 1, 1, 2, 3, 5, 8, 13, ... donde los primeros dos términos son 0 y 1, y cada término a partir de entonces es la suma de los dos términos precedentes; es decir $Fib[n] = Fib[n - 1] + Fib[n - 2]$. Usando esta información, escriba un programa en C++ que calcule el *enésimo* número en una secuencia de Fibonacci, donde n sea introducido de manera interactiva en el programa por el usuario. Por ejemplo si $n = 6$, el programa deberá desplegar el valor 5.

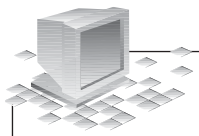
5.5 TÉCNICAS DE PROGRAMACIÓN CON CICLOS

En esta sección se presentan cuatro técnicas de programación comunes asociadas con ciclos de prueba preliminar (`for` y `while`). Todas estas técnicas son de uso común para los programadores experimentados.

Técnica 1: Entrada interactiva dentro de un ciclo

En la sección 5.2 se presentó el efecto de incluir una instrucción `cin` dentro de un ciclo `while`. Introducir datos en forma interactiva dentro de un ciclo es una técnica general que es aplicable por igual a los ciclos `for`. Por ejemplo, en el programa 5.12 se usa una instrucción `cin` para permitir a un usuario introducir de manera interactiva un conjunto de números. Conforme se introduce cada número, es sumado a un total. Cuando se sale del ciclo `for`, se calcula y se despliega el promedio.

La instrucción `for` en el programa 5.12 crea un ciclo que se ejecuta cuatro veces. Se le indica al usuario que introduzca un número cada vez que pasa el ciclo. Después que es introducido cada número, se suma de inmediato al total. Nótese que `total` es inicializado en cero como parte de la lista de inicialización que se ejecuta en la instrucción `for`. El ciclo en el programa 5.12 es ejecutado en tanto el valor en `cuenta` es menor que o igual a cuatro y es terminado cuando `cuenta` se vuelve cinco (el incremento a cinco, de hecho, es lo que causa que termine el ciclo). La salida producida por el programa 5.12 es en esencia la misma que en el programa 5.7.



Programa 5.12

```
#include <iostream>
using namespace std;

// Este programa calcula el promedio de CUENTAMAX
// numeros introducidos por el usuario
int main()
{

  const int CUENTAMAX = 4;
  int cuenta;
  double num, total, promedio;

  total = 0.0;

  for (cuenta = 0; cuenta < CUENTAMAX; cuenta++)
  {
    cout << "Introduzca un número: ";
    cin  >> num;
    total = total + num;
  }

  promedio = total / CUENTAMAX;
  cout << "El promedio de los datos introducidos es "
       << promedio << endl;
  return 0;
}
```

Técnica 2: Selección dentro de un ciclo

Otra técnica de programación común es usar un ciclo `for` o `while` para hacer un ciclo a través de un conjunto de números y seleccionar aquellos números que satisfagan uno o más criterios. Por ejemplo, suponga que desea encontrar la suma positiva y negativa de un conjunto de números. Los criterios aquí son si el número es positivo o negativo, y la lógica para poner en práctica este programa está dada por el seudocódigo

```
Mientras la condición del ciclo sea verdadera
    Introducir un número
     Si el número es mayor que cero
       sumar el número a la suma positiva
     De lo contrario
       sumar el número a la suma negativa
    Endif
Endwhile
```

El programa 5.13 describe este algoritmo en C++ para un ciclo de cuenta fija donde se han de introducir cinco números.

Programa 5.13

```
#include <iostream>
using namespace std;

// Este programa calcula las sumas positiva y negativa de un conjunto
// de numeros introducidos por el usuario en NUMSMAX
int main()
{
  const int NUMSMAX = 5;
  int i;
  double usenum, totpos, totneg;

  totpos = 0; // esta inicializacion puede hacerse en la declaracion
  totneg = 0; // esta inicializacion puede hacerse en la declaracion

  for (i = 1; i <= NUMSMAX; i++)
  {
    cout << "Introduzca un numero (positivo o negativo) : ";
    cin  >> usenum;
    if (usenum > 0)
      totpos = totpos + usenum;
    else
      totneg = totneg + usenum;
   }
   cout << "El total positivo es " << totpos << endl;
   cout << "El total negativo es " << totneg << endl;

   return 0;
}
```

La siguiente es una muestra de ejecución usando el programa 5.13.

```
Introduzca un numero (positivo o negativo) : 10
Introduzca un numero (positivo o negativo) : -10
Introduzca un numero (positivo o negativo) : 5
Introduzca un numero (positivo o negativo) : -7
Introduzca un numero (positivo o negativo) : 11
El total positivo es 26
El total negativo es -17
```

Técnica 3: Evaluación de funciones de una variable

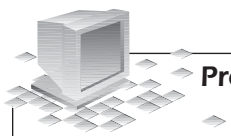
Los ciclos pueden construirse convenientemente para determinar y desplegar los valores de una sola variable de función matemática para un conjunto de valores sobre cualquier intervalo especificado. Por ejemplo, suponga que desea conocer los valores de la función

$$y = 10x^2 + 3x - 2$$

para x entre 2 y 6. Suponiendo que x ha sido declarada como una variable de número entero, puede utilizarse el siguiente ciclo `for` para calcular los valores requeridos.

```
for (x = 2; x <= 6; x++)
{
  y = 10 * pow(x,2) + 3 * x - 2;
  cout << setw(4) << x
       << setw(11) << y << endl;
}
```

Para este ciclo se ha usado la variable `x` como la variable del contador y como la incógnita (variable independiente) en la función. Para cada valor de `x` de dos a cinco se calcula y se despliega un nuevo valor de `y`. Este ciclo `for` está contenido dentro del programa 5.14, el cual también despliega encabezados apropiados para los valores impresos.



Programa 5.14

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  int x, y;

  cout << "valor x   valor y\n"
       << "-------   --------\n"
  for (x = 2; x <= 6; x++)
  {
    y = 10 * pow(x,2) + 3 * x - 2;
    cout << setw(4) << x
         << setw(11) << y << endl;
  }

  return 0;
}
```

Cuando se ejecuta el programa 5.14 se despliega lo siguiente:

```
valor x    valor y
-------    --------
   2          44
   3          97
   4         170
   5         263
   6         376
```

Aquí son de importancia dos elementos. El primero es que cualquier ecuación con una incógnita puede evaluarse usando un solo ciclo `for` o un ciclo `while` equivalente. El método requiere sustituir la ecuación deseada en el ciclo en lugar de la ecuación usada en el programa 5.14, y ajustar los valores del contador para que correspondan con el rango de solución deseado.

El segundo elemento que se debe considerar es que no estamos restringidos a usar valores enteros para la variable del contador. Por ejemplo, al especificar un incremento no entero, pueden obtenerse soluciones para valores fraccionarios. Esto se muestra en el programa 5.15, donde la ecuación $y = 10x^2 + 3x - 2$ es evaluada en el rango $x = 2$ a $x = 6$ en incrementos de 0.5.

Programa 5.15

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  double x, y;

  cout << "valor x     valor y\n";
       << "-------     ---------\n"
  cout << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(5);
  for (x = 2.0; x <= 6.0; x = x + 0.5)
  {
    y = 10.0 * pow(x,2.0) + 3.0 * x - 2.0;
    cout << setw(7) <<  x
         << setw(14) << y << endl;
  }

  return 0;
}
```

Hay que observar que *x* y *y* han sido declaradas como variables de punto flotante en el programa 5.15 para permitir que estas variables adopten valores fraccionarios. La siguiente es la salida producida por este programa.

```
valor x      valor y
-------     ---------
2.00000      44.00000
2.50000      68.00000
3.00000      97.00000
3.50000     131.00000
4.00000     170.00000
4.50000     214.00000
5.00000     263.00000
5.50000     317.00000
6.00000     376.00000
```

Técnica 4: Control interactivo de un ciclo

Los valores usados para controlar un ciclo pueden establecerse usando variables en lugar de valores constantes. Por ejemplo, las cuatro instrucciones

```
i = 5;
j = 10;
k = 1;
for (cuenta = i; cuenta <= j; cuenta = cuenta + k)
```

producen el mismo efecto que la instrucción única

```
for (count = 5; count <= 10; count = count + 1)
```

Del mismo modo, las instrucciones

```
i = 5;
j = 10;
k = 1;
cuenta = i;
while (cuenta <= j)
  cuenta = cuenta + k;
```

producen el mismo efecto que el siguiente ciclo `while`

```
cuenta = 5;
while (cuenta <= 10)
  cuenta = cuenta + 1;
```

La ventaja de usar variables en las expresiones de inicialización, condición y alteración es que permiten asignar valores para estas expresiones que son externos a la instrucción `for`

o `while`. Esto es útil en especial cuando se usa una instrucción `cin` para establecer los valores reales. Para hacer esto un poco más tangible, considere el programa 5.16.

Programa 5.16

```
#include <iostream>
#include <iomanip>
using namespace std;

// este programa despliega una tabla de numeros, sus cuadrados y cubos
// empezando por el numero 1. El numero final en la tabla es
// introducido por el usuario

int main()
{
  int num, final;

  cout << "Introduzca el numero final para la tabla: ";
  cin  >> final;

  cout << "NUMERO CUADRADO CUBO\n";
  cout << "------ -------- ----\n";

  for (num = 1; num <= final; num++)
    cout << setw(3) << num
         << setw(8) << num*num
         << setw(7) << num*num*num << endl;

  return 0;
}
```

En el programa 5.16, se ha usado una variable para controlar la expresión de condición (la de en medio). Aquí se ha colocado una instrucción `cin` antes del ciclo para permitir al usuario decidir cuál debería ser el valor final. Nótese que este arreglo permite al usuario establecer el tamaño de la tabla en tiempo de ejecución, en lugar de hacer que el programador establezca el tamaño de la tabla en tiempo de compilación. Esto también hace que el programa sea más general, en virtud que ahora puede usarse para crear una variedad de tablas sin la necesidad de reprogramar y volver a compilar.

Ejercicios 5.5

1. **`cin` dentro de un ciclo:** Escriba y ejecute un programa en C++ que acepte seis temperaturas Fahrenheit, una a la vez, y convierta cada valor introducido en su equivalente Celsius antes que se solicite el siguiente valor. Use un ciclo `for` en su programa. La conversión requerida es *Celsius* = (5.0/9.0) · (*Fahrenheit* – 32).

2. **`cin` dentro de un ciclo:** Escriba y ejecute un programa en C++ que acepte 10 valores individuales de galones, uno a la vez, y convierta cada valor introducido a su equivalente en litros antes que se solicite el siguiente valor. Use un ciclo `for` en su programa. Use el hecho que hay 3.785 litros en un galón.

3. **Control interactivo del ciclo:** Modifique el programa escrito para el ejercicio 2 para solicitar inicialmente el número de datos que se introducirán y convertirán.

4. **Control interactivo del ciclo:** Modifique el programa 5.13 de modo que el número de entradas que va a haber sea especificado por el usuario cuando se ejecute el programa.

5. **Selección:** Modifique el programa 5.13 de modo que despliegue el promedio de los números positivos y negativos. (*Sugerencia*: Tenga cuidado de no contar el número cero como un número negativo.) Pruebe su programa introduciendo los números 17, –10, 19, 0, –4. El promedio positivo desplegado por su programa deberá ser 18 y el promedio negativo, –7.

6. **a. Selección:** Escriba un programa en C++ que seleccione y despliegue el valor máximo de cinco números que se introducirán cuando el programa se ejecute. (*Sugerencia:* Use un ciclo `for` con una instrucción `cin` y una `if` internas en el ciclo.)
   b. Modifique el programa escrito para el ejercicio 6a de modo que despliegue tanto el valor máximo como la posición en el conjunto de números introducido donde ocurre el máximo.

7. **Selección:** Escriba un programa en C++ que seleccione y despliegue los primeros 20 números enteros que sean divisibles entre 3.

8. **Selección:** Los padres de una niña le prometieron darle 10 dólares cuando cumpliera 12 años de edad y duplicar el regalo en cada cumpleaños subsiguiente hasta que el regalo excediera 1000 dólares. Escriba un programa en C++ para determinar qué edad tendrá la niña cuando se le dé la última cantidad y la cantidad total recibida.

9. **Funciones matemáticas:** Modifique el programa 5.15 para producir una tabla de valores y para lo siguiente:
   a. $y = 3x^5 - 2x^3 + x$
   para x entre 5 y 10 en incrementos de 0.2
   b. $y = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}$
   para x entre 1 y 3 en incrementos de 0.1
   c. $y = 2e^{0.8t}$ para t entre 4 y 10 en incrementos de 0.2

10. **Funciones matemáticas:** Un modelo de población mundial, en miles de millones de personas, está dado por la ecuación

    $Población = 6.0e^{0.02t}$)

    donde t es el tiempo en años ($t = 0$ representa enero de 2000 y $t = 1$ representa enero de 2001). Usando esta fórmula, escriba un programa en C++ que despliegue una tabla de población anual para los años de enero de 2005 a enero de 2010.

11. **Funciones matemáticas:** Las coordenadas x y y como una función del tiempo, t, de un proyectil lanzado con una velocidad inicial v en un ángulo de θ con respecto al suelo están dadas por:

    $x = v\ t$ coseno(θ)
    $y = v\ t$ seno(θ)

    Usando estas fórmulas, escriba un programa en C++ que despliegue una tabla de valores de x y y de y para un proyectil lanzado con una velocidad inicial de 500 pies/s en un ángulo de 22.8 grados. (*Sugerencia:* Recuerde convertir la medida a radianes.) La tabla deberá contener valores correspondientes al intervalo de tiempo de 0 a 10 segundos en incrementos de $\frac{1}{2}$ segundo.

5.6 CICLOS ANIDADOS

En muchas situaciones es conveniente usar un ciclo contenido dentro de otro ciclo. Estos ciclos se llaman **ciclos anidados**. Un ejemplo simple de un ciclo anidado es

```
for(i = 1; i <= 5; i++)      // inicio del ciclo exterior
{
  cout << "\ni es ahora " << i << endl;

  for(j = 1; j <= 4; j++)    // inicio del ciclo interior
    cout << "  j = " << j;   // fin del ciclo interior
}                            // fin del ciclo exterior
```

El primer ciclo, controlado por el valor de `i`, se llama *ciclo exterior*. El segundo ciclo, controlado por el valor de `j`, se llama *ciclo interior*. Hay que observar que todas las instrucciones en el ciclo interior están contenidas dentro de los límites del ciclo exterior y que se ha usado una variable diferente para controlar cada ciclo. En cada ciclo del ciclo exterior, el ciclo interior recorre su secuencia por completo. Por tanto, cada vez que el contador `i` aumenta en 1, el ciclo `for` interior se ejecuta por completo. Esta situación se ilustra en la figura 5.9.

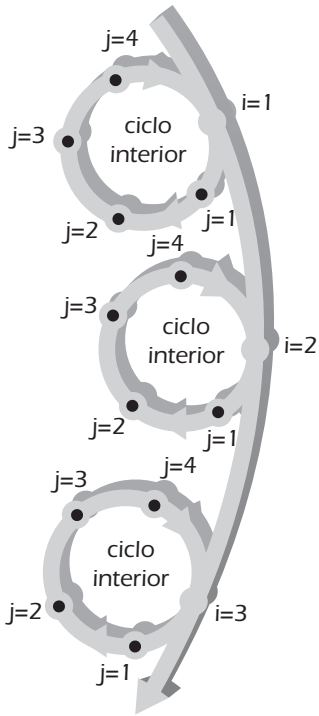


Figura 5.9 Ciclos para cada `i`, `j`.

El programa 5.17 incluye este tipo de código en un programa funcional.

Un ejemplo de salida del programa 5.17 es

```
i es ahora 1
  j = 1  j = 2  j = 3  j = 4
i es ahora 2
  j = 1  j = 2  j = 3  j = 4
i es ahora 3
  j = 1  j = 2  j = 3  j = 4
i es ahora 4
  j = 1  j = 2  j = 3  j = 4
i es ahora 5
  j = 1  j = 2  j = 3  j = 4
```

Programa 5.17

```
#include <iostream>
using namespace std;

int main()
{
  const int MAXI = 5;
  const int MAXJ = 4;
  int i, j;

  for(i = 1; i <= MAXI; i++)     // inicio del ciclo exterior <----+
  {                              //                                |
    cout << "\ni es ahora " << i << endl;  //                      |
                                 //                                |
    for(j = 1; j <= MAXJ; j++)   // inicio del ciclo interior      |
      cout << "  j = " << j;     // fin del ciclo interior         |
  }                              // fin del ciclo exterior   <-----+

  cout << endl;

  return 0;
}
```

Para ilustrar la utilidad de un ciclo anidado, se usará uno para calcular la calificación promedio para cada estudiante en una clase de 20 estudiantes. Cada estudiante ha presentado cuatro exámenes durante el semestre. La calificación final se calcula como el promedio de las calificaciones en estos exámenes. El seudocódigo que describe cómo puede hacerse este cálculo es

```
For 20 veces
   Establecer el total de calificaciones del estudiante en cero
   For 4 veces
      introducir una calificación
      sumar la calificación al total
   Endfor  // fin del ciclo for interior
   Calcular el promedio de calificaciones del estudiante
   Imprimir el promedio de calificaciones del estudiante
Endfor  // fin del ciclo for exterior
```

Como describe el seudocódigo, se usará un ciclo exterior consistente en 20 ciclos para calcular el promedio de calificaciones para cada estudiante. El ciclo interior consiste de cuatro ciclos. La calificación de un examen se introduce en cada pasada del ciclo interior. Conforme se introduce cada calificación se suma al total para el estudiante, y al final del ciclo se calcu-

la y se despliega el promedio. En vista que tanto el ciclo exterior como el interior son ciclos de cuenta fija de 20 y 4, respectivamente, se usarán instrucciones `for` para crear estos ciclos. El programa 5.18 proporciona el código C++ correspondiente al seudocódigo.

Programa 5.18

```
#include <iostream>
using namespace std;

int main()
{

  const int NUMCALIF = 4;
  const int NUMESTUDIANTES = 20;
  int i, j;
  double calificacion, total, promedio;

  for (i = 1; i <= NUMESTUDIANTES; i++) // inicio del ciclo exterior
  {
    total = 0;                        // limpia el total para este estudiante
    for (j = 1; j <= NUMCALIF; j++) // inicio del ciclo interior
    {
      cout << "Introduzca una calificacion de examen para este estudiante: ";
      cin  >> calificacion;
      total = total + calificacion;     // suma la calificacion en el total
    }                                   // fin del ciclo for interior
    promedio = total / NUMCALIF;        // calcula el promedio
    cout << "\nEl promedio para el estudiante " << i
         << " es " << promedio << "\n\n";
  }                                     // fin del ciclo for exterior

  return 0;
}
```

Al revisar el programa 5.18, ponga particular atención en la inicialización de total dentro del ciclo exterior, antes que se introduzca el ciclo interior. `total` se inicializa 20 veces, una para cada estudiante. También hay que observar que el promedio se calcula y despliega inmediatamente después de terminado el ciclo interior. En vista que las instrucciones que calculan e imprimen el promedio también están contenidas dentro del ciclo exterior, se calculan y despliegan 20 promedios. La introducción y adición de cada calificación dentro del ciclo interior utilizan técnicas que ya se habían visto antes, con las cuales ya debe estar familiarizado ahora.

Ejercicios 5.6

1. Se llevaron a cabo cuatro experimentos, cada uno consistente en seis resultados de prueba. Los resultados para cada experimento se dan a continuación. Escriba un programa usando un ciclo anidado para calcular y desplegar el promedio de los resultados de prueba para cada experimento.

   Resultados del primer experimento: 23.2 31 16.9 27 25.4 28.6
   Resultados del segundo experimento: 34.8 45.2 27.9 36.8 33.4 39.4
   Resultados del tercer experimento: 19.4 16.8 10.2 20.8 18.9 13.4
   Resultados del cuarto experimento: 36.9 39 49.2 45.1 42.7 50.6

2. **a.** Modifique el programa escrito para el ejercicio 1 de modo que el número de resultados de prueba para cada experimento sea introducido por el usuario. Escriba su programa de modo que pueda introducirse un número diferente de resultados de prueba para cada experimento.
   b. Vuelva a escribir el programa escrito para el ejercicio 2a para eliminar el ciclo interior.

3. **a.** Un fabricante de equipo eléctrico prueba cinco generadores midiendo sus voltajes de salida en tres momentos diferentes. Escriba un programa en C++ que use un ciclo anidado para introducir los resultados de prueba de cada generador y luego calcule y despliegue el voltaje promedio para cada generador. Suponga los siguientes resultados de prueba de los generadores:

   Primer generador: 122.5 122.7 123.0
   Segundo generador: 120.2 127.0 125.1
   Tercer generador: 121.7 124.9 126.0
   Cuarto generador: 122.9 123.8 126.7
   Quinto generador: 121.5 124.7 122.6

   b. Modifique el programa escrito para el ejercicio 3a para calcular y desplegar el voltaje promedio para todos los generadores. (*Sugerencia:* Use una segunda variable para almacenar el total de todos los voltajes de los generadores.)

4. Vuelva a escribir el programa escrito para el ejercicio 3a para eliminar el ciclo interior. Para hacer esto, tendrá que introducir tres voltajes para cada generador en lugar de uno a la vez. Cada voltaje debe almacenarse en su propio nombre de variable antes que se calcule el promedio.

5. Escriba un programa que calcule y despliegue valores para y cuando

   $$y = xz/(x - z)$$

   Su programa deberá calcular y para valores de x que varían entre 1 y 5 y valores de z que varían entre 2 y 6. La variable x deberá controlar el ciclo exterior e incrementarse en pasos de 1 y z deberá incrementarse en pasos de 1. Su programa deberá desplegar también el mensaje `Funcion Indefinida` cuando los valores de x y de z sean iguales.

6. Los lenguajes ensambladores para algunos microprocesadores no tienen una operación de multiplicación. Aunque hay algoritmos sofisticados para llevar a cabo la multiplicación en estos casos, un método simple multiplica por adición repetida. En este caso la eficiencia del algoritmo puede incrementarse usando ciclos anidados. Por

ejemplo, para multiplicar un número por doce, primero suma el número tres veces y luego suma el resultado cuatro veces. Esto sólo requiere siete adiciones en vez de doce. Usando esta información escriba un programa en C++ que multiplique 33, 47 y 83 por 1001 usando tres ciclos y luego despliegue el resultado. (*Sugerencia:* 1001 = 7 · 11 · 13)

5.7 Ciclos do while

Tanto las instrucciones `while` como las `for` evalúan una expresión al inicio del ciclo de repetición; como tales siempre se usan para crear ciclos de prueba preliminar. Los ciclos de prueba posterior, los cuales también se conocen como ciclos controlados a la salida, pueden construirse de igual forma en C++. La estructura básica de un ciclo así, el cual se conoce como ciclo `do while`, se ilustra en la figura 5.10. Nótese que un ciclo do `while` continúa las iteraciones a través del ciclo mientras la condición es verdadera y sale del ciclo cuando la condición es falsa.

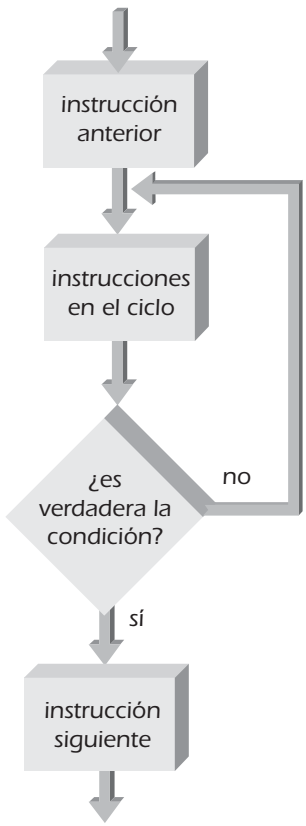


Figura 5.10 Estructura del ciclo `do while`.

En C++, un ciclo `do while` de prueba posterior se crea usando una instrucción `do`. Como su nombre implica, esta instrucción permite hacer algunas instrucciones antes que sea evaluada una expresión al final del ciclo. La forma general de la instrucción `do` de C++ es

```
do
  instrucción;
while (expresión);
```

no olvide el final `;`

Como con todos los programas en C++, la instrucción única en `do` puede reemplazarse por una instrucción compuesta. En la figura 5.11 se muestra un diagrama de control de flujo que ilustra la operación de la instrucción `do`.

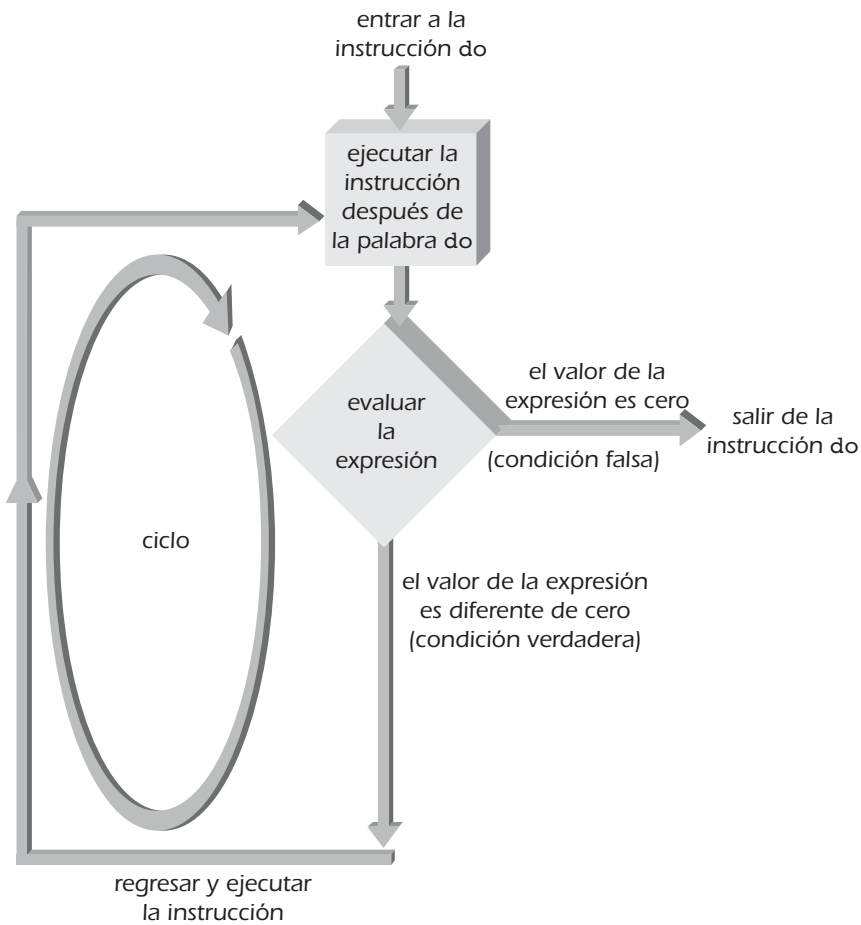


Figura 5.11 Flujo de control de la instrucción `do`.

Como se ilustró, todas las instrucciones dentro de la instrucción do son ejecutadas al menos una vez antes que la expresión sea evaluada. Luego, si la expresión tiene un valor diferente de cero, las instrucciones se ejecutan de nuevo. Este proceso continúa hasta que la expresión se evalúa en cero (se vuelve falsa). Por ejemplo, considere la siguiente instrucción do:

```
do
{
  cout << "\nIntroduzca un precio: ";
  cin  >> precio;
  if (abs(precio - CENTINELA) < 0.0001)
    break;
  impuestoventa = TASA * precio;
  cout << setiosflags(ios::showpoint)
       << setprecision(2)
       << "El impuesto sobre la venta es $ " << impuestoventa;
}
while (precio != CENTINELA);
```

Obsérvese que el indicador y la instrucción cin se incluyen dentro del ciclo debido a que la expresión probada es evaluada al final del ciclo.

Como todas las instrucciones de repetición, la instrucción do siempre puede reemplazar o ser reemplazada por una instrucción while o for equivalente. La elección de cuál instrucción usar depende de la aplicación y el estilo preferido por el programador. En general, se prefieren las instrucciones while y for porque permiten que cualquiera que lea el programa sepa con claridad lo que se está probando "por adelantado" en la parte superior del ciclo del programa.

Verificaciones de validez

La instrucción do es útil en particular para filtrar datos introducidos por el usuario y proporcionar verificaciones de validación de los datos. Por ejemplo, suponga que se requiere un operador para introducir un número de identificación del cliente válido entre los números 1000 y 1999. Un número que esté fuera de este rango se rechazará y se hará una nueva solicitud para un número válido. La siguiente sección de código proporciona el filtrado de datos necesario para verificar la introducción de un número de identificación válido:

```
do
{
  cout << "\nIntroduzca un numero de identificacion: ";
  cin  >> num_id;
}
while (num_id < 1000 || num_id > 1999);
```

Aquí, una solicitud para la introducción de un número de identificación se repite hasta que se introduce un número válido. Esta sección de código es "el meollo" ya que ni alerta al operador de la causa de la nueva solicitud de datos ni permite la salida prematura del ciclo si no puede encontrar un número de identificación válido. Una alternativa para eliminar el primer inconveniente es

```
do
{
  cout << "\nIntroduzca un numero de identificacion: ";
  cin  >> num_id;
  if (num_id < 1000 || num_id > 1999)
  {
    cout << "Se acaba de introducir un numero invalido\n";
    cout << "Por favor verifique el numero de identificacion
             y vuelva a introducirlo\n";
  }
  else
    break;   // interrumpe si se introdujo un numero de
                identificacion valido
} while(1);  // esta expresion siempre es verdadera
```

Aquí se ha usado una instrucción `break` para salir del ciclo. En vista que la expresión que está evaluando la instrucción `do` siempre es 1 (verdadera), se ha creado un ciclo infinito que sólo se interrumpe cuando se encuentra la instrucción `break`.

Ejercicios 5.7

1. **a.** Usando una instrucción `do`, escriba un programa para aceptar una calificación. El programa deberá solicitar una calificación en forma continua en tanto se introduzca una calificación inválida. Una calificación inválida es cualquier calificación menor que 0 o mayor que 100. Después que se ha introducido una calificación válida, su programa deberá desplegar el valor de la calificación introducida.
   b. Modifique el programa escrito para el ejercicio 1a de modo que el usuario sea alertado cuando se ha introducido una calificación inválida.
   c. Modifique el programa escrito para el ejercicio 1b de modo que permita al usuario salir del programa introduciendo el número 999.
   d. Modifique el programa escrito para el ejercicio 1b de modo que termine en forma automática después que se han introducido cinco calificaciones inválidas.
2. **a.** Escriba un programa que solicite en forma continua que se introduzca una calificación. Si la calificación es menor que 0 o mayor que 100, su programa deberá imprimir un mensaje apropiado que informe al usuario que se ha introducido una calificación inválida, de lo contrario la calificación deberá sumarse a un total. Cuando se introduzca una calificación de 999 el programa deberá salir del ciclo de repetición y calcular y desplegar el promedio de las calificaciones válidas introducidas.
   b. Ejecute el programa escrito en el ejercicio 2a en una computadora y verifique el programa usando datos de prueba apropiados.
3. **a.** Escriba un programa para invertir los dígitos de un número entero positivo. Por ejemplo, si se introduce el número 8735, el número desplegado deberá ser 5378. (*Sugerencia:* Use una instrucción do y continuamente quite y despliegue el dígito de las unidades del número. Si la variable `num` en un inicio contiene el número introducido, el dígito de las unidades se obtiene como (`num % 10`). Después que se despliega un dígito de unidades, dividir el número entre 10 establece el número para la siguiente iteración. Por tanto (8735 % 10) es 5 y (8735 / 10) es 873. La instrucción `do` deberá continuar en tanto el número remanente no sea cero.

b. Ejecute el programa escrito en el ejercicio 3a en una computadora y verifique el programa usando datos de prueba apropiados.

4. Repita cualquiera de los ejercicios en la sección 5.3 usando una instrucción `do` en lugar de una instrucción `for`.

5. Dado un número *n*, y una aproximación para su raíz cuadrada, puede obtenerse una aproximación más cercana a la raíz cuadrada real usando la fórmula:

$$\textit{aproximación nueva} = \frac{(n \,/\, \textit{aproximación previa}) + \textit{aproximación previa}}{2}$$

Usando esta información, escriba un programa en C++ que indique al usuario que introduzca un número y una estimación inicial de su raíz cuadrada. Usando estos datos de entrada su programa deberá calcular una aproximación a la raíz cuadrada que tenga una precisión hasta 0.00001 (*Sugerencia:* Detenga el ciclo cuando la diferencia entre dos aproximaciones sea menor que 0.00001.)

6. Aquí hay un problema desafiante para aquellos que saben un poco de cálculo. El método de Newton-Raphson puede utilizarse para encontrar las raíces de cualquier ecuación $y(x) = 0$. En este método la $(i + 1)$*ésima* aproximación, x_{i+1}, a una raíz de $y(x) = 0$ está dada en términos de la iésima aproximación, x_i, por la fórmula

$$x_{i+1} = x_i - y(x_i) \,/\, y'(x_i)$$

Por ejemplo, si $y(x) = 3x^2 + 2x - 2$, entonces $y'(x) = 6x + 2$, y las raíces se encuentran haciendo una estimación razonable para una primera aproximación x_1 y haciendo iteraciones usando la ecuación

$$x_{i+1} = x_i - (3x_i^2 + 2x_i - 2) \,/\, (6x_i + 2)$$

a. Usando el método de Newton-Raphson, encuentre las dos raíces de la ecuación $3x^2 + 2x - 2 = 0$. (*Sugerencia:* Hay una raíz positiva y una raíz negativa.)

b. Extienda el programa escrito para el ejercicio 6a de modo que encuentre las raíces de cualquier función $y(x) = 0$, cuando la función para $y(x)$ y la derivada de $y(x)$ son colocadas en el código.

5.8 Errores comunes de programación

Por lo general los programadores en C++ principiantes cometen seis errores cuando usan instrucciones de repetición. El más problemático de éstos es el error de "fallar por uno", donde el ciclo se ejecuta ya sea una vez más o una vez menos de lo que se pretendía. Por ejemplo, el ciclo creado por la instrucción `for(i = 1; i < 11; i++)` se ejecuta diez veces, no once, aun cuando se use el número 11 en la instrucción. Por tanto, un ciclo equivalente puede construirse usando la instrucción `for(i = 1; i <= 10; i ++)`. Sin embargo, si el ciclo empieza con un valor inicial de i = 0, usando la instrucción `for(i = 0; i < 11; i++)`, el ciclo se repetirá 11 veces, al igual que un ciclo construido con la instrucción `for (i = 0; i <= 10; i++)`. Por tanto, al construir ciclos, debe ponerse atención particular a las condiciones inicial y final usadas para controlar el ciclo para asegurar que el número de repeticiones no falle por una ejecución de más o una de menos.

Los siguientes dos errores se relacionan con la expresión probada, y ya se han encontrado con las instrucciones `if` y `switch`. El primero es el uso inadvertido del operador de asignación, =, por el operador de igualdad, ==, en la expresión probada. Un ejemplo de este error es mecanografiar la expresión de asignación a = 5 en lugar de la expresión relacional deseada a==5. En vista que la expresión probada puede ser cualquier expresión válida en C++, incluyendo expresiones aritméticas y de asignación, este error no es detectado por el compilador.

Como con la instrucción `if`, las instrucciones de repetición no deben usar el operador de igualdad, ==, cuando prueben operandos de punto flotante o de precisión doble. Por ejemplo, la expresión `fnum == 0.01` deberá reemplazarse por una prueba que requiera que el valor absoluto de `fnum - 0.01` sea menor que una cantidad aceptable. La razón para esto es que todos los números son almacenados en forma binaria. Usando un número finito de bits, los números decimales como 0.01 no tienen un equivalente binario exacto, así que las pruebas que requieren igualdad con dichos números pueden fallar.

Los siguientes dos errores son particulares de la instrucción `for`. El más común es colocar un punto y coma al final de los paréntesis de `for`, lo cual con frecuencia produce un ciclo `no hacer nada`. Por ejemplo, considere las instrucciones

```
for(cuenta = 0; cuenta < 10; cuenta++);
  total = total + num;
```

Aquí el punto y coma al final de la primera línea de código es una instrucción nula. Esto tiene el efecto de crear un ciclo que es ejecutado 10 veces sin hacer nada excepto el incremento y prueba de `cuenta`. Este error tiende a ocurrir debido a que los programadores en C++ están acostumbrados a terminar la mayor parte de las líneas con un punto y coma.

El siguiente error ocurre cuando se usan comas para separar los elementos en una instrucción `for` en lugar de los puntos y comas requeridos. Un ejemplo de esto es la instrucción

```
for (cuenta = 1, cuenta < 10, cuenta++)
```

Las comas deben usarse para separar elementos dentro de las listas de inicialización y de alteración, pero los puntos y comas deben usarse para separar estas listas de la expresión probada.

El último error ocurre cuando se omite el punto y coma final de la instrucción `do`. Este error por lo general es cometido por programadores que han aprendido a omitir el punto y coma después del paréntesis de una instrucción `while` y conservan este hábito cuando encuentran la palabra reservada `while` al final de una instrucción `do`.

5.9 RESUMEN DEL CAPÍTULO

1. Una sección de código repetitivo se conoce como un *ciclo*.

El ciclo es controlado por una instrucción de repetición que prueba una condición para determinar si se ejecutará el código. Cada una de estas pruebas a través del ciclo se conoce como una *repetición* o *iteración*. La condición probada siempre debe establecerse de manera explícita antes de su primera evaluación por la instrucción de repetición. Dentro del ciclo siempre debe haber una instrucción que permita la alteración de la condición de modo que pueda salirse del ciclo, una vez comenzado.

2. Hay tres tipos básicos de ciclo:

   a. `while`

   b. `for`

   c. `do while`

   Los ciclos `while` y `for` son ciclos de prueba *preliminar* o *controlados en la entrada*. En este tipo de ciclo la condición probada se evalúa al principio del ciclo, el cual
   requiere que la condición probada se establezca de manera explícita antes de la entrada al ciclo. Si la condición es verdadera, comienzan las repeticiones del ciclo; de lo contrario no se entra al ciclo. Las iteraciones continúan en tanto la condición permanece verdadera. En C++, los ciclos `while` y `for` se construyen usando instrucciones `while` y `for`, respectivamente.

   El ciclo `do while` es un ciclo de prueba posterior o *controlado en la salida*, donde la condición probada se evalúa al final del ciclo. Este tipo de ciclo siempre se ejecuta al menos una vez. En tanto la condición probada permanezca verdadera, los ciclos `do while` continúan ejecutándose.

3. Los ciclos también se clasifican según el tipo de condición probada. En un *ciclo de cuenta fija*, la condición se usa para dar seguimiento a cuantas repeticiones han ocurrido. En un ciclo de *condición variable* la condición probada se basa en una variable que puede cambiar de manera interactiva con cada pasada a través del ciclo.

4. En C++, un ciclo `while` se construye usando una instrucción `while`. La forma más utilizada de esta instrucción es

   while (expresión)
   {
   instrucciones;
   }

   La expresión contenida dentro del paréntesis es la condición probada para determinar si se ejecuta la instrucción que sigue al paréntesis, la cual por lo general es una instrucción compuesta. La expresión es evaluada exactamente de la misma manera que si estuviera contenida en una instrucción `if-else`; la diferencia es cómo se usa la expresión. En una instrucción `while` la instrucción que sigue a la expresión es ejecutada de manera repetida en tanto la expresión mantenga un valor diferente de cero, en lugar de sólo una vez, como en una instrucción `if-else`. Un ejemplo de un ciclo `while` es

```
cuenta = 1;              // inicializar cuenta
while (cuenta <= 10)
{
  cout << cuenta << "  ";
  cuenta++;                 // incrementar cuenta
}
```

   La primera instrucción de asignación establece cuenta igual a 1. Luego se introduce la instrucción `while` y la expresión es evaluada por primera vez. En vista que el valor de cuenta es menor que o igual a 10, la expresión es verdadera y la instrucción compuesta se ejecuta. La primera instrucción en la instrucción compuesta usa el

objeto `cout` para desplegar el valor de cuenta. La siguiente instrucción agrega 1 al valor almacenado en la actualidad en cuenta, haciendo este valor igual a 2. La instrucción `while` regresa ahora a probar de nuevo la expresión. En vista que `cuenta` aún es menor que o igual a 10, la instrucción compuesta se ejecuta de nuevo. Este proceso continúa hasta que el valor de `cuenta` llega a 11.

La instrucción `while` siempre verifica esta expresión al principio del ciclo. Esto requiere que cualesquiera variables en la expresión probada deben tener valores asignados antes que se encuentre `while`. Dentro del ciclo `while` debe haber una instrucción que altere el valor de la expresión probada.

5. En C++, un ciclo `for` se construye usando una instrucción `for`. Esta instrucción ejecuta las mismas funciones que la instrucción `while`, pero utiliza una forma diferente. En muchas situaciones, en especial aquellas que usan una condición de cuenta fija, el formato de la instrucción `for` es más fácil de usar que su instrucción `while` equivalente. La forma más usada de la instrucción `for` es

```
for (lista de inicialización; expresión; lista de alteración)
{
  instrucciones;
}
```

Dentro del paréntesis de la instrucción `for` hay tres elementos, separados por puntos y comas. Cada uno de estos elementos es opcional pero los puntos y comas deben estar presentes.

La lista de inicialización se usa para establecer los valores iniciales antes de entrar al ciclo; por lo general se usa para inicializar un contador. Las instrucciones dentro de la lista de inicialización sólo se ejecutan una vez. La expresión en la instrucción `for` es la condición que se prueba al inicio del ciclo y antes de cada iteración. La lista de alteración contiene instrucciones de ciclo que no están contenidas dentro de la instrucción compuesta: por lo general se usa para incrementar o disminuir un contador cada vez que se ejecuta el ciclo. Las instrucciones múltiples dentro de una lista se separan con comas. Un ejemplo de un ciclo `for` es

```
for (total = 0, cuenta = 1; cuenta < 10; cuenta++)
{
  cout << "Introduzca una calificacion: ";
  total = total + calificacion;
}
```

En esta instrucción `for`, la lista de inicialización se usa para inicializar tanto total como cuenta. La expresión determina que el ciclo se ejecutará en tanto el valor en cuenta sea menor que 10, y el valor de cuenta se incrementa en uno cada vez que se pasa por el ciclo.

6. La instrucción `for` es útil en extremo para crear ciclos de cuenta fija. Esto se debe a que las instrucciones de inicialización, la expresión probada y las instrucciones que afectan a la expresión probada pueden incluirse en el paréntesis al inicio de un ciclo `for` para una fácil inspección y modificación.

7. La instrucción `do` se usa para crear ciclos de prueba posterior porque verifica su expresión al final del ciclo. Esto asegura que el cuerpo de un ciclo `do` se ejecute al menos una vez. Dentro de un ciclo `do` debe haber al menos una instrucción que altere el valor de la expresión probada.

Consideración de opciones de carrera

Ingeniería industrial

Cada una de las disciplinas de la ingeniería tradicional (civil, mecánica, eléctrica, química y metalúrgica/minera) se fundamenta en un área particular de las ciencias naturales. La ingeniería industrial, sin embargo, incorpora el conocimiento de las ciencias sociales al diseño de mejoras en sistemas hombre-máquina. Los ingenieros industriales son responsables del diseño, instalación y evaluación de máquinas y sistemas y también de la supervisión de su interfaz con las personas para mejorar la productividad general. Este trabajo puede implicar la comprensión de las características conductuales humanas y sus efectos en el diseño de máquinas o del lugar de trabajo. Los ingenieros industriales hacen mucho énfasis en el conocimiento de economía, administración de negocios y finanzas, al igual que en las ciencias naturales. Las áreas de especialización del ingeniero industrial pueden dividirse en cuatro categorías:

1. Investigación de operaciones. Esta área implica la aplicación de técnicas analíticas y modelos matemáticos a fenómenos como control de inventarios, simulación, teoría de las decisiónes y teoría de colas para optimizar los sistemas totales necesarios para la producción de bienes.
2. Ingeniería administrativa. La interacción cada vez más compleja de la administración con las habilidades de producción en las operaciones industriales modernas ha generado una gran demanda de gerentes con capacitación técnica. Estos gerentes evalúan y planean empresas corporativas e interactúan con la fuerza laboral, los departamentos de ingeniería y los subcontratistas. Un ingeniero administrativo también puede participar en las operaciones financieras de una compañía, debido a sus conocimientos de economía, administración de negocios y leyes.
3. Ingeniería de manufactura y producción. Antes que se fabrique un producto, el proceso completo de manufactura debe ser diseñado e instalado para optimizar la economía implicada y la calidad final del artículo. Esta tarea requiere un conocimiento amplio de diseño de procesos, trazado de plantas, diseño de herramientas, robótica e interacciones hombre-máquina.
4. Sistemas de información. Esta área implica el uso de computadoras para recopilar y analizar datos para la toma de decisiones y la planeación y para mejorar la actividad hombre-máquina.

La siguiente lista incluye las responsabilidades más comunes de los ingenieros industriales que respondieron a una encuesta reciente realizada por el Instituto Estadounidense de Ingenieros Industriales:

Planeación y diseño de instalaciones
Ingeniería de métodos
Diseño de sistemas de trabajo
Ingeniería de producción
Sistemas de información y control administrativo
Análisis y diseño de organizaciones
Medición del trabajo
Administración de salarios
Control de calidad
Administración de proyectos
Control de costos
Control de inventarios
Conservación de energía
Control de procesos computarizados
Empaque, manejo y prueba de productos
Selección de herramientas y equipo
Control de producción
Estudio del mejoramiento de productos
Mantenimiento preventivo
Programas de seguridad
Programas de capacitación

CAPÍTULO 6
Modularidad con el uso de funciones

TEMAS

6.1 DECLARACIONES DE FUNCIONES Y PARÁMETROS
- PROTOTIPOS DE FUNCIÓN
- LLAMADA A UNA FUNCIÓN
- DEFINICIÓN DE UNA FUNCIÓN
- COLOCACIÓN DE INSTRUCCIONES
- CABOS DE FUNCIÓN
- FUNCIONES CON LISTAS DE PARÁMETROS VACÍAS
- ARGUMENTOS POR OMISIÓN
- REUTILIZACIÓN DE NOMBRES DE FUNCIÓN (SOBRECARGA)
- PLANTILLAS DE FUNCIÓN

6.2 DEVOLVER UN SOLO VALOR
- FUNCIONES `inline`

6.3 DEVOLVER VALORES MÚLTIPLES
- TRANSMISIÓN Y USO DE PARÁMETROS DE REFERENCIA

6.4 APLICACIONES
- APLICACIÓN 1: CONVERSIÓN DE COORDENADAS RECTANGULARES A POLARES
- APLICACIÓN 2: SIMULACIÓN

6.5 ALCANCE DE UNA VARIABLE
- OPERADOR DE RESOLUCIÓN DE ALCANCE
- MAL USO DE LAS VARIABLES GLOBALES

6.6 CLASES DE ALMACENAMIENTO DE VARIABLES
- CLASES DE ALMACENAMIENTO DE VARIABLES LOCALES
- CLASES DE ALMACENAMIENTO DE VARIABLES GLOBALES

6.7 ERRORES COMUNES DE PROGRAMACIÓN

6.8 RESUMEN DEL CAPÍTULO
- CONSIDERACIÓN DE OPCIONES DE CARRERA: INGENIERÍA QUÍMICA

Los programas profesionales se diseñan, codifican y prueban en forma muy parecida al hardware: como una serie de módulos que están integrados para funcionar en conjunto. Una buena analogía de esto es un automóvil en el cual un módulo fundamental es el motor, otro es la transmisión, un tercero el sistema de frenos, un cuarto la carrocería, etc. Cada uno de estos módulos está vinculado entre sí y colocado al final bajo el control del conductor, el cual puede compararse con un supervisor o un módulo principal de un programa. El conjunto opera ahora como una unidad completa, capaz de hacer trabajo útil, como llevarnos a la tienda. Durante el proceso de montaje, cada módulo se construye, prueba y se encuentra libre de defectos en forma individual antes de ser instalado en el producto final.

Ahora piense en lo que podría hacer si quisiera mejorar el desempeño de su automóvil. Podría alterar el motor existente o eliminarlo por completo y armar un motor nuevo. Del mismo modo, podría cambiar la transmisión o las llantas o los amortiguadores, haciendo cada modificación en forma individual conforme se lo permitan su tiempo y su presupuesto. En cada caso, la mayor parte de los otros módulos pueden permanecer igual, pero el automóvil ahora opera en forma diferente.

En esta analogía, cada uno de los componentes principales de un automóvil puede compararse con una función. Por ejemplo, el conductor llama al motor cuando se oprime el pedal del acelerador. El motor acepta entradas de combustible, aire y electricidad para convertir la solicitud del conductor en un producto útil, potencia, y luego envía esta salida a la transmisión para su procesamiento adicional. La transmisión recibe la salida del motor y la convierte en una forma que pueda ser usada por el eje de transmisión. Una entrada adicional a la transmisión es la selección de velocidades por parte del conductor (primera, reversa, neutral, etcétera).

En cada caso, el motor, la transmisión y otros módulos sólo "conocen" el universo limitado por sus entradas y salidas. El conductor no necesita saber nada de la operación interna del motor, transmisión, aire acondicionado y otros módulos que se están controlando. Todo lo que requiere es comprender qué hace cada unidad y cómo usarla. El conductor tan sólo "llama" a un módulo, como el motor, los frenos, el aire acondicionado y la dirección cuando se requiere la salida de ese módulo. La comunicación entre módulos está restringida a pasar las entradas necesarias a cada módulo conforme se le llame a ejecutar su tarea, y cada módulo opera en forma interna de una manera relativamente independiente. Este mismo enfoque modular es usado por los ingenieros para crear y mantener programas confiables en C++ usando funciones.

Como se ha visto, cada programa en C++ debe contener una función `main()`*. Además de esta función requerida, los programas en C++ también pueden contener cualquier cantidad de funciones adicionales. En este capítulo se aprenderá cómo escribir estas funciones, transmitirles datos, procesar los datos transmitidos y devolver un resultado.*

6.1 DECLARACIONES DE FUNCIONES Y PARÁMETROS

Al crear funciones en C++, debemos poner atención a la función en sí y en la forma en que interactúa con otras funciones, como `main()`. Esto incluye transmitir datos en forma correcta a una función cuando es invocada y devolver valores de una función. En esta sección se describe la primera parte de la interfaz, transmitir datos a una función y hacer que la función reciba, almacene y procese en forma correcta los datos transmitidos.

Como ya se ha visto con las funciones matemáticas, una función se invoca, o utiliza, dando el nombre de la función y transmitiéndole datos, como argumentos, en el paréntesis que sigue al nombre de la función (véase la figura 6.1).

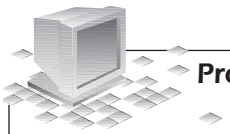
nombre-de-la-función (*datos transmitidos a la función*)

Esto identifica a la función llamada — Esto transmite datos a la función

Figura 6.1 Llamar y transmitir datos a una función.

La función invocada debe ser capaz de aceptar los datos que le son transmitidos por la función que hace la llamada. Sólo después que la función invocada recibe con éxito los datos pueden ser manipulados éstos para producir un resultado útil.

Para aclarar el proceso de envío y recepción de datos, considérese el programa 6.1, el cual invoca a una función nombrada `encontrarMax()`. El programa, como se muestra, no está completo todavía. Una vez que se escribe la función `encontrarMax()` y se incluye en el programa 6.1, el programa completado, consistente en las funciones `main()` y `encontrarMax()`, puede ser compilado y ejecutado.



Programa 6.1

```
#include <iostream>
using namespace std;

void encontrarMax(int, int); // la declaracion de la funcion (prototipo)

int main()
{
  int primernum, segundonum;

  cout << "\nIntroduzca un numero: ";
  cin >> primernum;
  cout << "¡Estupendo! Por favor introduzca un segundo numero: ";
  cin >> segundonum;

  encontrarMax(primernum, segundonum); // aquí se llama la funcion

  return 0;
}
```

Examinemos la declaración y llamada de la función `encontrarMax()` desde `main()`. Luego vamos a escribir `encontrarMax()` para aceptar los datos que se le transmiten y determinar el valor mayor o máximo de los dos valores transmitidos.

La función `encontrarMax()` se conoce como la ***función llamada,*** en vista que es llamada o invocada a la acción por su referencia en `main()`. La función que hace la llamada, en este caso `main()`, se conoce como la ***función que llama.*** Los términos llamada y que llama vienen del uso del teléfono estándar, donde una persona llama a otra. La persona que inicia la llamada se conoce como quien llama, y quien la recibe es la persona llamada. Los mismos términos describen las llamadas a la función. La función llamada, en este caso `encontrarMax()`, es declarada como una función que espera recibir dos números enteros y no devolver ningún valor (void) `a main()`. Esta declaración se conoce de manera formal como un prototipo de función. La función es invocada entonces por la última instrucción en el programa.

Prototipos de función

Antes que una función pueda ser llamada, debe ser declarada la función que hará la llamada. La instrucción de declaración para una función se conoce como un **prototipo de función.** El prototipo de función le indica a la función que llama el tipo de valor que será devuelto formalmente, si es que hay alguno, y el tipo de datos y orden de los valores que la función que llama deberá transmitir a la función llamada. Por ejemplo, el prototipo de función usado antes en el programa 6.1

```
void encontrarMax(int, int);
```

declara que la función `encontrarMax()` espera que se le envíen dos valores enteros, y que esta función particular devuelve de manera formal ningún valor (void). Los prototipos de función pueden colocarse con las instrucciones de declaración de variable de la función que llama, encima del nombre de la función que llama, como en el programa 6.1, o en un archivo de encabezado separado que se incluirá utilizando una instrucción de preprocesamiento `#include.` Por tanto, el prototipo de función para `encontrarMax()` podría haberse colocado antes o después de la instrucción `#include <iostream>,` antes de `main()` o dentro de `main()`. (Las razones para la elección de la colocación se presentan en la sección 6.3.) La forma general de las instrucciones de prototipo de función es:

tipo-de-datos-a-devolver nombre-de-función (lista de tipos de datos para los argumentos);

donde el tipo de datos se refiere al tipo del valor que será devuelto de manera formal por la función. Son ejemplos de prototipos de función

```
int fmax(int, int);
double intercambio(int, char, char, double);
void desplegar(double, double);
```

El prototipo de función para `fmax()` declara que esta función espera recibir dos argumentos en número entero y devolverá de manera formal un valor en número entero. El prototipo de función para `intercambio()` declara que esta función requiere cuatro argumentos consistentes en un número entero, dos caracteres y un argumento de precisión doble, en este orden y formalmente retornará un número de precisión doble. Por último, el prototipo de función para `desplegar()` declara que esta función requiere dos argumentos de precisión doble y no devuelve ningún valor. Dicha función podría ser utilizada para desplegar los resultados de un cálculo en forma directa, sin devolver ningún valor a la función llamada.

El uso de prototipos de función permite la verificación de errores en los tipos de datos por el compilador. Si el prototipo de función no concuerda con los tipos de datos definidos cuando se escribe la función, ocurrirá una advertencia del compilador. El prototipo también sirve para otra tarea: asegura la conversión de todos los argumentos transmitidos a la función al tipo de datos del argumento declarado cuando se llama la función.

Llamada a una función

Llamar a una función es una operación bastante fácil. Los únicos requisitos son usar el nombre de la función y que los datos transmitidos a la función estén encerrados dentro de los paréntesis que siguen al nombre de la función usando el mismo orden y tipo que se declaró en el prototipo de función. Los elementos encerrados dentro de los paréntesis se llaman **argumentos** de la función llamada (véase la figura 6.2).

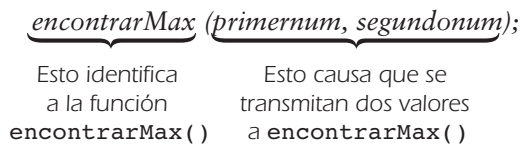


Figura 6.2 *Llamar a* `encontrarMax()` *y transmitirle dos valores.*

Si una variable es uno de los argumentos en una llamada la función, la función llamada recibe una copia del valor almacenado en la variable. Por ejemplo, la instrucción `encontrarMax(primernum, segundonum);` invoca a la función `encontrarMax` y causa que los valores que residen en la actualidad en las variables `primernum` y `segundonum` sean transmitidos a `encontrarMax()`. Los nombres de variable entre paréntesis son argumentos que proporcionan valores a la función llamada. Después que se transmiten los valores, el control es transferido a la función llamada.

Como se ilustra en la figura 6.3, la función `encontrarMax()` *no recibe las variables nombradas* `primernum` *y* `segundonum` *y no tiene conocimiento de estos nombres de variables.*[1] La función tan sólo recibe los valores en estas variables y debe determinar por sí misma dónde almacenarlos antes de hacer algo más. Aunque este procedimiento para transmitir datos a una función puede parecer sorprendente, en realidad es un procedimiento de seguridad para que una función invocada no cambie de manera inadvertida los datos almacenados en una variable. La función obtiene una copia de los datos que va a usar. Puede cambiar su copia y, por supuesto, cambiar cualesquiera variables declaradas dentro de sí misma. Sin embargo, a menos que se den pasos específicos para hacerlo, no se le permite a una función cambiar el contenido de las variables declaradas en otras funciones.

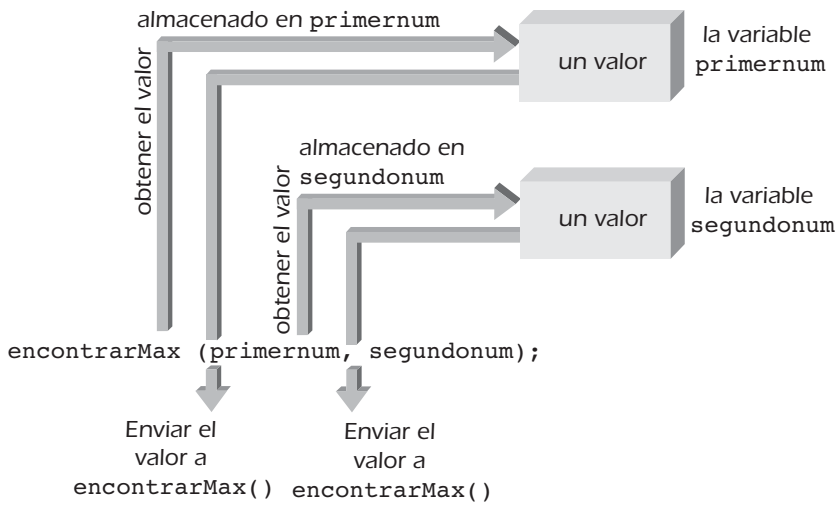


Figura 6.3 `encontrarMax()` *recibe valores actuales.*

Ahora comenzaremos a escribir la función `encontrarMax()` para procesar los valores que se le transmitieron.

[1] En la sección 6.3 se verá cómo, al usar variables de referencia, C++ permite también el acceso directo a las variables de la función que invoca.

Definición de una función

Una función se define cuando se escribe. Cada función es definida una vez (es decir, escrita una vez) en un programa y puede ser usada entonces por cualquier otra función en el programa que la declare en forma adecuada.

Como la función `main()`, toda función en C++ consta de dos partes, un **encabezado de función** y un **cuerpo de función**, como se ilustra en la figura 6.4. El propósito del encabezado de función es identificar el tipo de datos del valor devuelto por la función, proporcionarle un nombre a la función y especificar el número, orden y tipo de los argumentos esperados por ella. El propósito del cuerpo de función es operar sobre los datos transmitidos y devolver en forma directa, cuando mucho, un valor a la función que llama. (En la sección 6.3, se verá, cómo puede hacerse que una función devuelva múltiples valores.)

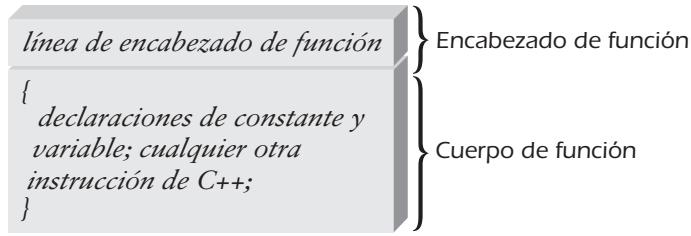


Figura 6.4 Formato general de una función.

El encabezado de función siempre es la primera línea de una función y contiene el tipo de valor devuelto de la función, su nombre y los nombres y tipos de datos de sus argumentos. En vista que `encontrarMax()` no devolverá de manera formal ningún valor y recibirá dos argumentos en número entero, puede usarse la siguiente línea de encabezado:

```
void encontrarMax(int x, int y)  <-- sin punto y coma
```

Los nombres de argumento en el encabezado se conocen como **parámetros formales** de la función.[2] Por tanto, el parámetro x se usará para almacenar el primer valor transmitido a `encontrarMax()` y el parámetro y se utilizará para almacenar el segundo valor transmitido en el momento de la llamada a la función. La función no sabe de dónde provienen los valores cuando se hace la llamada desde `main()`. La primera parte del procedimiento de llamada ejecutado por la computadora implica ir a las variables `primernum` y `segundonum` y recuperar los valores almacenados. Estos valores son transmitidos luego a `encontrarMax()` y almacenados al final en los parámetros `x` y `y` (véase la figura 6.5).

[2] La porción del encabezado de función que contiene el nombre de la función y los parámetros se conoce de manera formal como un *declarador de función*.

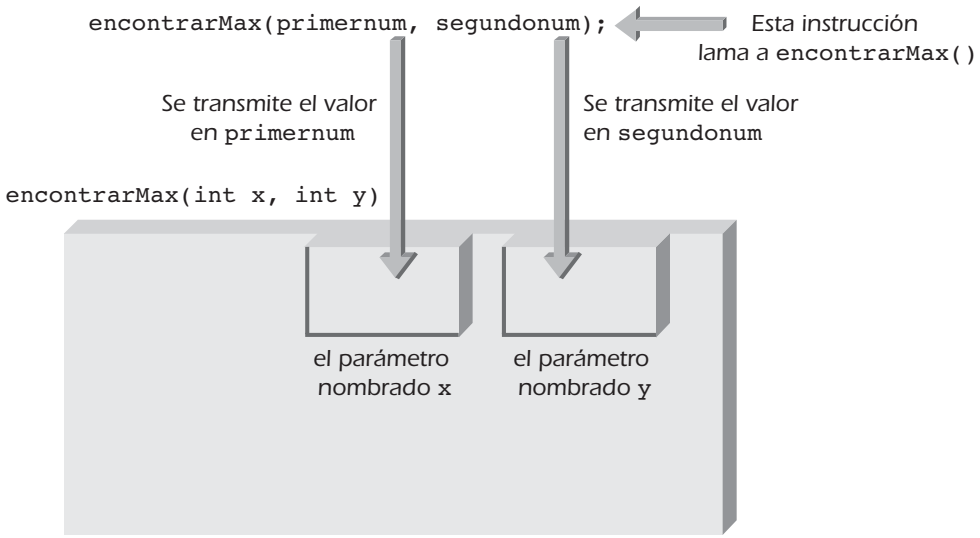


Figura 6.5 *Almacenar valores en parámetros.*

El nombre de la función y todos los nombres de parámetros en el encabezado, en este caso `encontrarMax, x` y `y`, son elegidos por el programador. Pueden usarse los nombres seleccionados de acuerdo con las reglas usadas para elegir nombres de variables. Todos los parámetros enumerados en la línea de encabezado de la función deben separarse con comas y deben tener los tipos de datos individuales declarados por separado.

Ahora que hemos escrito el encabezado de función para la función `encontrarMax(),` se puede construir su cuerpo. Supongamos que la función `encontrarMax()` selecciona y despliega el más grande de los dos números que se le transmiten.

Como se ilustra en la figura 6.6, un cuerpo de función comienza con una llave de apertura, `{,` contiene las declaraciones necesarias y otras instrucciones de C++, y termina con una llave de cierre, `}.` Esto debería serle familiar porque es la misma estructura usada en todas las funciones `main()` que se han escrito. Esto no debería ser sorpresa porque `main()` en sí es una función y debe apegarse a las reglas requeridas para construir todas las funciones legítimas.

```
{
  declaraciones de constantes simbólicas,
  declaraciones de variables y otras
  instrucciones de C++
}
```

Figura 6.6 Estructura de un cuerpo de función.

Punto de Información

Definiciones de función y prototipos de función

Cuando se escribe una función, se crea de manera formal una definición de función. Cada definición comienza con una línea de encabezado que incluye una lista de parámetros, si es que hay alguna, encerrada entre paréntesis y termina con la llave de cierre que finaliza el cuerpo de función. Los paréntesis se requieren use o no algún parámetro la función. Una sintaxis utilizada por lo común para una definición de función es:

```
tipo-de-datos-que-devuelve nombre-de-funcion (lista de parámetros)
{
  declaraciones de constantes
  declaraciones de variables

  otras instrucciones de C++

  return valor
}
```

Un **prototipo de función** declara una función. La sintaxis de un prototipo de función, el cual proporciona el tipo de datos que devuelve la función, el nombre de la función y la lista de argumentos de la función es:

```
tipo-de-datos-que-devuelve nombre-de-funcion (lista de tipos de
datos de los parámetros);
```

Como tal, el prototipo junto con comentarios previos y posteriores a la condición (véase el siguiente recuadro Punto de información) deberán proporcionar al usuario toda la información de programación necesaria para invocar a la función con éxito.

En general, todos los prototipos de funciones se colocan al principio del programa, y todas las definiciones se colocan después de la función `main()`. Sin embargo, esta colocación puede cambiarse. El único requisito en C++ es que una función no puede ser llamada antes que haya sido declarada o definida.

En el cuerpo de la función `encontrarMax()`, se declarará una variable para almacenar el máximo de los dos números transmitidos a ella. Entonces se usará una instrucción `if-else` para encontrar el máximo de los dos números. Por último, se usará una instrucción `cout` para desplegar el máximo. La definición de función completa para la función `encontrarMax()` es:

```
void encontrarMax(int x, int y)
{                    // inicio del cuerpo de funcion
  int numMax;        // declaración de variable

  if (x >= y)        // find the maximum number
    numMax = x;
  else
    numMax = y;

  cout << "\nEl máximo de los dos números es "
       << numMax << endl;

}  // fin del cuerpo de funcion y fin de la funcion
```

Punto de información

Punto de información

Las condiciones previas son cualquier conjunto de condiciones que una función requiere que sean verdaderas para operar en forma correcta. Por ejemplo, si una función usa la constante simbólica `MAXCHARS`, la cual debe tener un valor positivo, una condición previa es que `MAXCHARS` sea declarado con un valor positivo antes que la función sea invocada.

Del mismo modo, una condición posterior es una condición que será verdadera después que se ejecute la función, suponiendo que se cumplen las condiciones previas.

Las condiciones previas y posteriores se documentan de manera típica como comentarios al usuario. Por ejemplo, considere la siguiente declaración y los comentarios:

```
bool bisiesto(int)
// Condiciones previas: los numeros enteros deben representar un año en un formato
//              : de cuatro digitos, como 2006
// Condiciones posteriores: debe devolver un valor de verdadero si el año
                            es bisiesto;
//              : de lo contrario se devolvera falso
```

Los comentarios de las condiciones previas y posteriores deberán incluirse tanto en los prototipos de función como en las definiciones de función siempre que se necesiten aclaraciones.

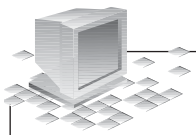
Nótese que las declaraciones de parámetros se hacen dentro de la línea de encabezado y la declaración de variables se hace inmediatamente después de la llave de apertura del cuerpo de función. Ello está de acuerdo con el concepto que los valores de parámetro son transmitidos a una función desde fuera de la función, y que las variables son declaradas y se les asignan valores desde dentro del cuerpo de función.

El programa 6.2 incluye la función `encontrarMax()` dentro del código del programa enlistado antes en el programa 6.1.

El programa 6.2 puede usarse para seleccionar e imprimir el máximo de dos números enteros cualesquiera introducidos por el usuario. A continuación se ve una muestra de la ejecución del programa 6.2:

```
Introduzca un número: 25
¡Estupendo! Por favor introduzca un segundo número: 5

El máximo de los dos números es 25
```

La colocación de la función `encontrarMax()` después de la función `main()` en el programa 6.2 es cuestión de elección. Por lo general enlistaremos primero `main()` porque es la función conductora que le dará a cualquiera que lea el programa una idea de lo que se refiere el programa completo antes de encontrar los detalles de cada función. Sin embargo, en ningún caso puede colocarse la definición de `encontrarMax()` dentro de `main()`. Esto es verdad para todas las funciones de C++, las cuales deben definirse por sí mismas fuera de cualquier otra función. Cada función en C++ es una entidad separada e independiente con sus propios parámetros y variables; *nunca se permite el anidamiento de funciones*.



Programa 6.2

```
#include <iostream>
using namespace std;

void encontrarMax(int, int); // el prototipo de la función

int main()
{
  int primernum, segundonum;

  cout << "\nIntroduzca un número: ";
  cin >> primernum;
  ccout << "¡Estupendo! Por favor introduzca un segundo numero: ";
  cin >> segundonum;

  encontrarMax(primernum, segundonum); // aquí se llama a la función

  return 0;
}

// en seguida está la función encontrarMax()

void encontrarMax(int x, int y)
{                   // inicio del cuerpo de función
  int numMax;       // declaración de variable

  if (x >= y)       // encontrar el número máximo
    numMax = x;
  else
    numMax = y;

  cout << "\nEl máximo de los dos números es "
       << numMax << endl;

  return;
}  // fin del cuerpo de función y fin de la función
```

Colocación de instrucciones

C++ no impone una estructura de ordenamiento de instrucciones rígida al programador. La regla general para colocar instrucciones en un programa en C++ es tan sólo que todas las directivas del preprocesador, constantes nombradas, variables y funciones deben declararse o definirse antes que puedan usarse. Como se ha señalado antes, aunque esta regla permite que tanto las directivas del preprocesador como las instrucciones de declaración se coloquen a través del programa, hacerlo así produce una estructura de programa muy pobre.

Como una cuestión de buena forma de programación, el siguiente ordenamiento de instrucciones deberá formar la estructura básica alrededor de la cual se construyan todos sus programas en C++.

```
directivas del preprocesador

prototipos de función

int main()
{
  constantes simbólicas
  declaraciones de variables

  otras instrucciones ejecutables

  return valor
}

definiciones de función
```

Como siempre, pueden entremezclarse comentarios a las instrucciones en cualquier parte dentro de esta estructura básica.

Cabos de función

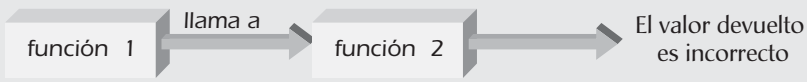
Una alternativa para completar cada función requerida en un programa completo es escribir primero la función `main()`, y agregar las funciones más adelante, conforme se desarrollan. El problema que surge con este enfoque, sin embargo, es el mismo que ocurrió con el programa 6.1; es decir, el programa no puede ejecutarse hasta que todas las funciones se han incluido. Por conveniencia a continuación se ha reproducido el código para el programa 6.1.

```
#include <iostream>
using namespace std;

void encontrarMax(int, int); // la declaración de la función
(prototipo)

int main()
{
  int primernum, segundonum;

  cout << "\nIntroduzca un número: ";
  cin >> primernum;
  cout << "¡Estupendo! Por favor introduzca un segundo número: ";
  cin >> segundonum;

  encontrarMax(primernum, segundonum); // aquí se llama la fúnción

  return 0;
}
```

Punto de Información

Prueba de aislamiento

Uno de los métodos más exitosos que se conoce para probar software es incorporar siempre el código que se está probando dentro de un ambiente de código funcional. Por ejemplo, suponga que tiene dos funciones no probadas que son llamadas en el orden que se muestra a continuación, y el resultado devuelto por la segunda función es incorrecto.



A partir de la información mostrada en esta figura, una de las funciones o quizás ambas podrían estar operando en forma incorrecta. El primer paso para solucionar el problema es aislar una función específica.

Uno de los métodos más potentes para llevar a cabo este aislamiento de código es desacoplar las funciones. Esto se hace probando cada función en forma individual o probando una función primero y, sólo cuando se sabe que está operando en forma correcta, reconectarla a la segunda función. Luego, si ocurre un error, lo habrá aislado a la transferencia de datos entre funciones o a la operación interna de la segunda función.

Este procedimiento específico es un ejemplo de la *regla básica de la prueba,* la cual establece que cada función sólo debería probarse en un programa en el que se sabe que todas las demás funciones son correctas. Esto significa que una función debe probarse primero por sí misma, usando cabos si es necesario para las funciones llamadas, y una segunda función debería probarse ya sea por sí misma o con una función probada con anterioridad, y así en forma sucesiva. Esto asegura que cada función nueva es aislada dentro de una capa de prueba de funciones correctas, con el programa final de código de funciones probadas construido de manera efectiva.

Este programa estaría completo si existiera una definición de función para `encontrarMax.` Pero en realidad no se necesita una función `encontrarMax` *correcta* para probar y ejecutar lo que se ha escrito, sólo necesitamos una función que *actúe* como si fuera una `encontrarMax` "falsa" que acepte el número y tipos apropiados de parámetros y devuelva valores de la manera apropiada para invocar la función es todo lo que se necesita para permitir la prueba inicial. Esta función falsa se llama cabo. Un **cabo** es el comienzo de una función final que se puede usar como un sustituto de la unidad final hasta que la unidad esté completa. Un cabo para `encontrarMax` es como sigue:

```
void encontrarMax(int x, int y)
{
  cout << "En encontrarMax()\n";
  cout << "El valor de x es " << x << endl;
  cout << "El valor de x es " << y << endl;
}
```

Esta función cabo puede ahora compilarse y vincularse con el código completado con anterioridad para obtener un programa ejecutable. Entonces el código para la función puede desarrollarse más con el código "real" cuando es completado, reemplazando la porción cabo.

El requerimiento mínimo de una función cabo es que se compila y se vincula con el módulo que llama. En la práctica, es una buena idea hacer que un cabo despliegue un mensaje que se ha introducido con éxito y el valor o los valores de sus parámetros recibidos, como en el cabo para `encontrarMax().`

Conforme se depura la función, puede permitir que haya cada vez más, quizá devolver resultados intermedios o incompletos. Este refinamiento creciente, o por etapas, es un concepto importante en el desarrollo eficiente de programas que le proporcionan el medio para ejecutar un programa que todavía no satisface todos sus requerimientos finales.

Funciones con listas de parámetros vacías

Aunque las funciones útiles que tienen una lista de parámetros vacía son limitadas en extremo (se proporciona una de dichas funciones en el ejercicio 11), pueden ocurrir. El prototipo de función para dicha función requiere escribir la palabra clave void o no poner nada en absoluto entre los paréntesis que siguen al nombre de la función. Por ejemplo, ambos prototipos

```
int despliegue();
```

e

```
int despliegue(void);
```

indican que la función `despliegue()` no tiene parámetros y devuelve un número entero. Una función con una lista de parámetros vacía es llamada por su nombre sin nada escrito dentro del paréntesis requerido después del nombre de la función. Por ejemplo, la instrucción `despliegue();` llama en forma correcta a la función `despliegue()` cuyo prototipo se proporcionó antes.

Argumentos por omisión[3]

Una característica conveniente de C++ es su flexibilidad para proporcionar argumentos por omisión mediante una llamada a la función. El uso primario de los argumentos por omisión es extender la lista de parámetros de funciones existentes sin requerir algún cambio en las listas de argumentos invocados que ya están en su lugar dentro de un programa.

Los valores del argumento por omisión se enlistan en el prototipo de función y son transmitidos en forma automática a la función llamada cuando los argumentos correspondientes son omitidos de la llamada a la función. Por ejemplo, el prototipo de función

```
void ejemplo (int, int = 5, double = 6.78);
```

proporciona valores por omisión para los últimos dos argumentos. Si alguno de estos argumentos se omite cuando la función es llamada en realidad, el compilador de C++ suministrará estos valores por omisión. Por tanto, todas las siguientes llamadas a la función son válidas:

```
ejemplo(7, 2, 9.3)  // no se usan valores por omisión
ejemplo(7, 2)       // igual que ejemplo(7, 2, 6.78)
ejemplo(7)          // igual que ejemplo(7, 5, 6.78)
```

Cuando se usan parámetros por omisión deben seguirse cuatro reglas. La primera es que los valores por omisión deberían asignarse en el prototipo de función.[4] La segunda es que si a cualquier parámetro se le da un valor por omisión en el prototipo de función, a todos los

[3] Este tema puede omitirse en la primera lectura sin perder la continuidad de la materia.

[4] Algunos compiladores aceptan asignaciones por omisión en la definición de la función.

parámetros que siguen también deben asignárseles valores por omisión. La tercera es que si un argumento se omite en la llamada a la función real, entonces todos los argumentos a su derecha también deben omitirse. Estas dos reglas le dejan claro de C++ cuáles argumentos se están omitiendo y le permite suministrar valores por omisión correctos para los argumentos faltantes, empezando por el argumento de la extrema derecha y trabajando hacia la izquierda. La última regla especifica que el valor por omisión utilizado en el prototipo de función puede ser una expresión consistente en constantes y variables declaradas con anterioridad. Si se usa una expresión así, debe pasar la verificación del compilador para variables declaradas con validez, aun cuando el valor real de la expresión sea evaluado y asignado en tiempo de ejecución.

Los argumentos por omisión son útiles en extremo cuando se extiende una función existente para incluir más características que requieren argumentos adicionales. Agregar los argumentos nuevos a la derecha de los argumentos existentes y proporcionarle a cada argumento nuevo un valor por omisión permite que todas las llamadas a funciones existentes permanezcan como están. Por tanto, el efecto de los cambios nuevos es aislado convenientemente del código existente en el programa.

Reutilización de nombres de función (sobrecarga)[5]

C++ proporciona la capacidad de usar el mismo nombre de función para más de una función, lo cual se conoce como **sobrecarga de función**. El único requisito para crear más de una función con el mismo nombre es que el compilador debe ser capaz de determinar cuál función usar con base en los tipos de datos de los parámetros (no los tipos de datos del valor devuelto, si es que hay alguno). Por ejemplo, considere las tres funciones siguientes, todas nombradas `cdabs().`

```
void cdabs(int x) // calcula y despliega el valor absoluto de un número entero
{
  if ( x < 0 )
    x = -x;
  cout << "El valor absoluto del número entero es " << x << endl;
}

void cdabs(float x) // calcula y despliega el valor absoluto de un número de
punto flotante
{
  if ( x < 0 )
    x = -x;
  cout << "El valor absoluto del número de punto flotante es " << x << endl;
}

void cdabs(double x) // calcula y despliega el valor absoluto de un número
en doble precision
{
  if ( x < 0 )
   x = -x;
  cout << "El valor absoluto del número de doble precisión es " << x << endl;
}
```

[5]Este tema puede omitirse en la primera lectura sin que pierda la continuidad de la materia.

Cuál de las tres funciones nombradas `cdabs()` se llama en realidad depende de los tipos de argumento suministrados en el momento de la llamada. Por tanto, la llamada a la función `cdabs(10);` causaría que el compilador utilizara la función nombrada `cdabs()` que espera un argumento de número entero, y la llamada a la función `cdabs(6.28f);` causaría que el compilador usara la función nombrada `cdabs()` que espera un argumento de precisión simple.[6]

Hay que observar que sobrecargar un nombre de función tan sólo significa usar el mismo nombre para más de una función. Cada función que utiliza el nombre debe escribirse aún y existe como una entidad separada. El uso del mismo nombre de función no requiere que el código dentro de las funciones sea similar, aunque la buena práctica de programación dicta que las funciones con el mismo nombre deberían ejecutar en esencia las mismas operaciones. Todo lo que se requiere de manera formal para usar el mismo nombre de función es que el compilador pueda distinguir cuál función seleccionar con base en los tipos de datos de los argumentos cuando es llamada la función. Sin embargo, es evidente que si todo lo que es diferente acerca de las funciones sobrecargadas son los tipos de argumentos, una mejor solución de programación es tan sólo crear una plantilla de función. El uso de funciones sobrecargadas, sin embargo, es útil en extremo con funciones constructoras, un tema que se presenta en la sección 8.3.

Plantillas de función[7]

En la mayor parte de los lenguajes de nivel alto, incluyendo C, el predecesor inmediato de C++, cada función requiere su propio nombre único. En teoría esto tiene sentido, pero en la práctica puede conducir a una profusión de nombres de función, incluso para funciones que realizan en esencia las mismas operaciones. Por ejemplo, considere determinar y desplegar el valor absoluto de un número. Si el número transmitido a la función puede ser un número entero, uno de precisión simple o un valor de precisión doble, deben escribirse tres funciones distintas para manejar en forma correcta cada caso. Por supuesto, podría dársele a cada una de estas funciones un nombre único, como `abs()`, `fabs()` y `dabs()`, respectivamente, teniendo los prototipos de función:

```
void abs(int);
void fabs(float);
void dabs(double);
```

Es evidente que cada una de estas funciones realiza en esencia la misma operación, pero con diferentes tipos de datos en los parámetros. Una solución mucho más limpia y elegante es escribir una función general que maneje todos los casos, pero cuyos parámetros, variables e incluso tipo devuelto puedan ser establecidos por el compilador basado en la llamada a la función real. Esto es posible en C++ usando plantillas de función.

[6] Esto se logra por un proceso conocido como *mutilación del nombre*. Usando este proceso el nombre de función generado en realidad por el compilador de C++ difiere del nombre de función usado en el código fuente. El compilador añade información al nombre de función del código fuente dependiendo del tipo de datos que se transmiten, y se dice que el nombre resultante es una versión mutilada del nombre del código fuente.

[7] Este tema puede omitirse en la primera lectura sin perder la continuidad de la materia.

Una **plantilla de función** es una función única completa que sirve como modelo para una familia de funciones. Cuál función de la familia se creará en realidad depende de las llamadas a la función subsiguientes. Para hacer esto más concreto, considere una plantilla de función que calcule y despliegue el valor absoluto de un argumento transmitido. Una plantilla de función apropiada es:

```
template <class T>
void mostrarabs(T numero)
{
  if (numero < 0)
    numero = -numero;
  cout << "El valor absoluto del numero "
       << " es " << numero << endl;

  return
}
```

Por el momento, ignore la primera línea `template <class T>` y observe la segunda línea, la cual consiste del encabezado de `función mostrarabs(T numero).` Hay que observar que esta línea de encabezado tiene la misma sintaxis que se ha usado para todas las definiciones de función, excepto por la `T` en el lugar donde se coloca por lo general un tipo de datos. Por ejemplo, si la línea de encabezado fuera `void mostrarabs(int numero),` lo reconocería como una función llamada `mostrarabs` que espera que se le transmita un argumento en número entero y que no devuelve ningún valor. Del mismo modo, si la línea de encabezado fuera `void mostrarabs(float numero),` la reconocería como una función que espera que se transmita un argumento de punto flotante cuando se llame a la función.

La ventaja al usar la `T` dentro de la línea de encabezado de la plantilla de función es que representa un tipo de datos general que es reemplazado por un tipo de datos real, como `int`, `float`, `double`, etc., cuando el compilador encuentra una llamada a la función real. Por ejemplo, si se encuentra una llamada a la función con un argumento en número entero, el compilador usará la plantilla de función para construir el código para una función que espera un parámetro en número entero. Del mismo modo, si se hace una llamada con un argumento de punto flotante, el compilador construirá una función que espera un parámetro en punto flotante. Como un ejemplo específico de esto, considérese el programa 6.3.

Primero observe las tres llamadas a función que se hacen en la función `main()` mostrada en el programa 6.3, la cual llama a la función `mostrarabs()` con un valor entero, flotante y doble, respectivamente. Ahora revise la plantilla de función para `mostrarabs()` y considere la primera línea `template <class T>.` Esta línea, la cual se llama **prefijo de plantilla,** se usa para informar al compilador que la función que sigue inmediatamente es una plantilla que usa un tipo de datos nombrado `T`. Dentro de la plantilla de función se usa la `T` de la misma manera que cualquier otro tipo de datos, como `int`, `float`, `double`, etc. Luego, cuando el compilador encuentra una llamada a la actual función para `mostrarabs()`, o el tipo de datos del argumento transmitido en la llamada es sustituido por `T` a lo largo de la función. En efecto, el compilador crea una función específica, usando la plantilla, que espera el tipo de argumento en la llamada. En vista que el programa 6.3 hace tres llamadas a `mostrarabs`, cada una con un tipo de datos diferente para el argumento, el compilador creará tres funciones `mostrarabs()` separadas. El compilador sabe cuál

Programa 6.3

```
#include <iostream>
using namespace std;

template <clases T>
void mostrarabs(T número)
{

  if (numero < 0)
    numero = -numero;
  cout << "El valor absoluto del número es "
       << numero << endl;

  return;
}

int main()
{
  int num1 = -4;
  float num2 = -4.23f;
  double num3 = -4.23456;

  mostrarabs(num1);
  mostrarabs(num2);
  mostrarabs(num3);

  return 0;
}
```

función usar con base en los argumentos transmitidos en el momento de la llamada. La salida desplegada cuando el programa 6.3 se ejecuta es:

```
El valor absoluto del numero es 4
El valor absoluto del numero es 4.23
El valor absoluto del numero es 4.23456
```

La letra `T` usada en el prefijo de plantilla `template <class T>` tan sólo es un sustituto para un tipo de datos que se define cuando la función es invocada en realidad. Puede

utilizarse en cambio cualquier letra o identificador que no sea una palabra clave. Por tanto, la plantilla de la función `mostrarabs()` podría haberse definido también como:

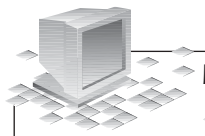
```
template <class TIPOD>
void abs(TIPOD numero)
{
  if (numero < 0)
    numero = -numero;
  cout << "El valor absoluto del número es "
       << numero << endl;

  return;
}
```

Así, en ocasiones es más simple y claro leer la palabra *clase* en el prefijo de plantilla que las palabras *tipo de datos*. Por tanto, el prefijo de plantilla `template <clases T>` puede leerse como "estamos definiendo una plantilla de función que tiene un tipo de datos nombrado `T`". Luego, dentro de la línea de encabezado y el cuerpo de la función definida se usa el tipo de datos `T` (o cualquier otra letra o identificador definido en el prefijo) de la misma manera que cualquier tipo de datos incorporado, como `int`, `float`, `double`, etcétera.

Ahora suponga que se quiere crear una plantilla de función para incluir el tipo de retorno y una variable declarada en forma interna. Por ejemplo, considere la siguiente plantilla de función:

```
template <clase T> // prefijo de plantilla
T abs(T valor) // linea de encabezado
{
  T numabs; // declaracion de variable

  if (valor < 0)
    numabs = -valor;
  else
    numabs = valor;

  return numabs;
}
```

En esta definición de plantilla, se ha usado el tipo de datos `T` para declarar tres elementos: el tipo devuelto de la función, el tipo de datos de un parámetro de función único llamado `valor`, y una variable declarada dentro de la función. El programa 6.4 ilustra cómo esta plantilla de función podría utilizarse dentro del contexto de un programa completo.



Programa 6.4

```
#include <iostream>
using namespace std;

template <class T> // prefijo de plantilla
T abs(T valor) // linea de encabezado
{
  T numabs; // declaracion de variable

  if (valor < 0)
    numabs = -valor;
  else
    numabs = valor;

  return numabs;
}
int main()
{
  int num1 = -4;
  float num2 = -4.23f;
  double num3 = -4.23456;

  cout << "El valor absoluto de " << num1
       << " es " << abs(num1) << endl;
  cout << "El valor absoluto de " << num2
       << " es " << abs(num2) << endl;
  cout << "El valor absoluto de " << num3
       << " es " << abs(num3) << endl;

  return 0;
}
```

En la primera llamada a `abs()` hecha dentro de `main()`, se transmite un valor entero como un argumento. En este caso, el compilador sustituye un tipo de datos `int` para el tipo de datos `T` en la plantilla de función y crea la siguiente función:

```
int abs(int valor) // linea de encabezado
{
  int numabs; // declaracion de la variable

  if (valor < 0)
    numabs = -valor;
  else
    numabs = valor;
```

```
  return numabs;
}
```

Del mismo modo, en la segunda y tercera llamadas a la función, el compilador crea dos funciones más, una en la que el tipo de datos `T` es reemplazado por la palabra clave `float`, y otra en la que el tipo de datos `T` es reemplazado por la palabra clave `double`. La salida producida por el programa 6.4 es:

```
El valor absoluto de -4 es 4
El valor absoluto de -4.23 es 4.23
El valor absoluto de -4.23456 es 4.23456
```

El valor de usar la plantilla de función es que se ha usado una definición de función para crear tres funciones diferentes, cada una de las cuales usa la misma lógica y operaciones pero opera sobre diferentes tipos de datos.

Por último, aunque los programas 6.3 y 6.4 definen una plantilla de función que usa un solo tipo de datos sustituto, pueden definirse plantillas de función con más de un tipo de datos. Por ejemplo, el prefijo de plantilla

```
template <class TIPOD1, class TIPOD2, class TIPOD3>
```

puede utilizarse para crear una plantilla de función que requiere tres tipos de datos diferentes. Como antes, dentro del encabezado y el cuerpo de la plantilla de función los tipos de datos `TIPOD1`, `TIPOD2` y `TIPOD3` se usarían de la misma manera que cualquier tipo de datos incorporado, como un `int`, `float`, `double`, etc. Además, como se señaló antes, los nombres `TIPOD1`, `TIPOD2` y `TIPOD3` pueden ser cualquier identificador que no sea una palabra clave. De manera convencional, se usaría la letra `T` seguida por cero o más dígitos, como `T`, `T1`, `T2`, `T3`, etcétera.

Ejercicios 6.1

1. Para los siguientes encabezados de función, determine el número, tipo y orden (secuencia) de los valores que deben transmitirse a la función:
   a. `void factorial(int n)`
   b. `void voltios(int res, double induc, double cap)`
   c. `void potencia(int tipo, double induc, double cap)`
   d. `void marcador(char tipo, double corriente, double tiempo)`
   e. `void total(double cantidad, double tasa)`
   f. `void roi(int a, int b, char c, char d, double e, double f)`
   g. `void obtener_valor(int elemento, int iter, char decmarcador, char delim)`

2. a. Escriba una función nombrada `revisar()` que tenga tres parámetros. El primer parámetro deberá aceptar un número entero, el segundo parámetro un número de precisión doble y el tercer parámetro un número de precisión doble. El cuerpo de la función deberá desplegar sólo los valores de los datos transmitidos a la función cuando es llamada. (*NOTA*: Cuando se rastrean errores en las funciones, es muy útil hacer que la función despliegue los valores que se le han transmitido. Con bastante frecuencia, el error no está en lo que el cuerpo de la función hace con los datos, sino en los datos recibidos y almacenados.)

b. Incluya la función escrita en el ejercicio 2a en un programa que funcione. Asegúrese que su función es llamada desde `main().` Pruebe la función transmitiéndole varios datos.

3. a. Escriba una función llamada `encontrarAbs()` que acepte un número trasmitido de precisión doble, calcule su valor absoluto y despliegue el valor absoluto. El valor absoluto de un número es el mismo número si éste es positivo y el negativo del número si éste es negativo.
b. Incluya la función escrita en el ejercicio 3a en un programa que funcione. Asegúrese que su función es llamada desde `main()`. Pruebe la función transmitiéndole varios datos.

4. a. Escriba una función llamada `mult()` que acepte dos números en punto flotante como parámetros, multiplique estos dos números y despliegue el resultado.
b. Incluya la función escrita en el ejercicio 4a en un programa que funcione. Asegúrese que su función es invocada desde `main()`. Pruebe la función transmitiéndole varios datos.

5. a. Escriba una función llamada al_`cuadrado()` que calcule el cuadrado del valor que se le transmite y despliegue el resultado. La función deberá ser capaz de elevar al cuadrado números con puntos decimales.
b. Incluya la función escrita en el ejercicio 5a en un programa que funcione. Asegúrese que su función es llamada desde `main()`. Pruebe la función transmitiéndole varios datos.

6. a. Escriba una función nombrada `funpot()` que eleve un número entero que se le transmita a una potencia en número entero positivo y despliegue el resultado. El número entero positivo deberá ser el segundo valor transmitido a la función. Declare la variable usada para almacenar el resultado como un tipo de datos entero largo para asegurar suficiente almacenamiento para el resultado.
b. Incluya la función escrita en el ejercicio 6a en un programa que funcione. Asegúrese que su función es llamada desde `main()`. Pruebe la función transmitiéndole varios datos.

7. a. Escriba un programa en C++ que devuelva la parte fraccionaria de cualquier número introducido por el usuario. Por ejemplo, si se introduce el número 256.879, debería desplegarse el número 0.879. (*Sugerencia:* Use un `int cast`.)
b. Introduzca, compile y ejecute el programa escrito para el ejercicio 7a.

8. a. Escriba un programa en C++ que acepte un argumento en número entero y determine si el entero transmitido es par o non. (*Sugerencia:* Utilice el operador %.)
b. Introduzca, compile y ejecute el programa escrito para el ejercicio 8a.

9. a. Escriba una función que produzca una tabla de los números del 1 al 10, sus cuadrados y sus cubos. La función deberá producir el mismo despliegue que el producido por el programa 5.11.
b. Incluya la función escrita en el ejercicio 9a en un programa que funcione. Asegúrese que su función es llamada desde `main()`. Pruebe la función transmitiéndole varios datos.

10. a. Modifique la función escrita para el ejercicio 9 para aceptar el valor inicial de la tabla, el número de valores que se van a desplegar y el incremento entre valores. Si el incremento no se envía de manera explícita, la función deberá usar un valor por omisión de 1. Nombre su función `selTab()`. Una llamada a `selTab(6, 5, 2);` deberá producir una tabla de cinco líneas, la primera línea iniciando con el número 6 y cada número subsiguiente incrementándose en 2.

b. Incluya la función escrita en el ejercicio 10a en un programa que funcione. Asegúrese que su función es llamada desde `main()`. Pruebe la función transmitiéndole varios datos.

11. Puede construirse una función útil que no utilice parámetros para que devuelva un valor para π que sea preciso al número máximo de lugares decimales permitido por su computadora. Este valor se obtiene tomando el arco seno de 1.0, el cual es π/2, y multiplicando el resultado por 2. En C++, la expresión requerida es 2.0* asin(1.0), donde la función `asin()` es proporcionada en la biblioteca matemática estándar de C++ (recuerde incluir `cmath`). Usando esta expresión, escriba una función en C++ llamada `pi()` que calcule y despliegue el valor de π.

12. a. Escriba una plantilla de función llamada `despliegue()` que despliegue el valor del argumento único que se le transmite cuando es invocada la función.

b. Incluya la plantilla de función creada en el ejercicio 12a dentro de un programa en C++ completo que llame a la función tres veces: una con un argumento de carácter, una con un argumento en número entero y una con un argumento de número de precisión doble.

13. a. Escriba una plantilla de función llamada `entero()` que devuelva el valor entero de cualquier argumento que se le transmita cuando la función sea llamada.

b. Incluya la plantilla de función creada en el ejercicio 13a dentro de un programa en C++ completo que llame a la función tres veces: una con un argumento de carácter, una con un argumento en número entero y una con un argumento de número de precisión doble.

14. a. Escriba una plantilla de función llamada `maximo()` que devuelva el valor máximo de tres argumentos que se transmitan a la función cuando sea llamada. Suponga que los tres argumentos serán del mismo tipo de datos.

b. Incluya la plantilla de función creada para el ejercicio 14a dentro de un programa en C++ completo que llame a la función con tres números enteros y luego con tres números de precisión doble.

15. a. Escriba una plantilla de función llamada al_`cuadrado()` que calcule y devuelva el cuadrado del argumento único transmitido a la función cuando es llamada.

b. Incluya la plantilla de función creada para el ejercicio 15a dentro de un programa en C++ completo.

6.2 DEVOLVER UN SOLO VALOR

Al utilizar el método de transmitir datos a una función presentado en la sección anterior, la función llamada sólo recibe copias de los valores contenidos en los argumentos en el momento de la llamada (repase la figura 6.3 si no le ha quedado claro esto). Cuando se transmite un valor a una función invocada de esta manera, el argumento transmitido se conoce como **transmitido por valor** y es una ventaja distintiva de C++.[8] En vista que la función llamada no tiene acceso directo a las variables usadas como argumentos por la función que llama, no puede alterar de manera inadvertida el valor almacenado en una de estas variables.

La función que recibe los argumentos transmitidos por valor puede procesar los valores que se le envían en cualquier forma deseada y devolver en forma directa cuando mucho uno, y sólo un, valor "legítimo" a la función que llama (véase la figura 6.7). En esta sección se verá cómo se devuelve un valor así a la función que llama. Como podría esperarse, dada la flexibilidad de C++, hay una forma de devolver más de un solo valor, pero éste es el tema de la siguiente sección.

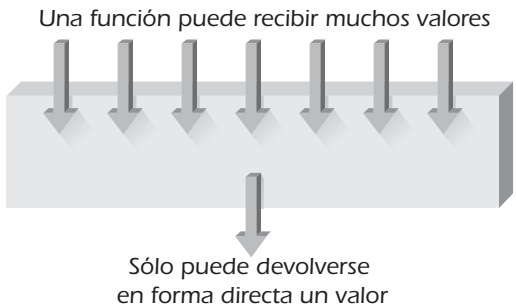


Figura 6.7 *Una función devuelve en forma directa cuando mucho un valor.*

Como ocurre con la llamada a una función, devolver en forma directa un valor requiere que la interfaz entre la función llamada y la que llama se maneje en forma correcta. Desde el lado de la transacción devuelta, la función llamada debe proporcionar los siguientes elementos:

- el tipo de datos del valor devuelto
- el valor real que se devuelve

Una función que devuelve un valor debe especificar, en su línea de encabezado, el tipo de datos del valor que se devolverá. Recuérdese que la línea de encabezado de la función es la primera línea de la función, la cual incluye tanto el nombre de la función como una lista de nombres de parámetros. Como ejemplo, considérese la función `encontrarMax()` escrita en la sección anterior. Determina el valor máximo de dos números transmitidos a la función. Por conveniencia, se enlista de nuevo el código `encontrarMax()`:

[8]Esto también se conoce como **llamada por valor.** Sin embargo, el término no se refiere a la llamada de la función en conjunto, sino a la forma en que se transmite un argumento individual cuando se hace la llamada a una función.

```
void encontrarMax(int x, int y)
{                     // inicio del cuerpo de la funcion
  int numMax;         // declaracion de la variable

  if (x >= y)         // encontrar el numero maximo
    numMax = x;
  else
    numMax = y;

  cout << "\nEl máximo de los dos números es "
       << numMax << endl;

}  // fin del cuerpo de la funcion y fin de la funcion
```

Tal como está escrito, la línea de encabezado de la función es

```
void encontrarMax(int x, int y)
```

donde `x` y `y` son los nombres elegidos para los parámetros de la función.

Si ahora `encontrarMax()` va a devolver un valor, la línea de encabezado de la función debe ser modificado para incluir el tipo de datos del valor que se devuelve. Por ejemplo, si se va a devolver un valor entero, la línea de encabezado de la función apropiada es

```
int encontrarMax(int x, int y)
```

Del mismo modo, si la función va a recibir dos parámetros de precisión simple y devolverá un valor de precisión simple, la línea de encabezado de la función correcta es

```
float encontrarMax(float x, float y)
```

y si la función va a recibir dos parámetros de precisión doble y a devolver un valor de precisión doble, la línea de encabezado sería[9]

```
double encontrarMax(double x, double y)
```

Ahora se modificará la función `encontrarMax()` para devolver el valor máximo de los dos números que se le transmiten. Para hacer esto, primero se debe determinar el tipo de datos del valor que se va a devolver e incluir este tipo de datos en la línea de encabezado de la función.

En vista que el valor máximo determinado por `encontrarMax()` se almacena en la variable de número entero `numMax`, es el valor de esta variable el que deberá devolver la función. Devolver un valor entero de `encontrarMax()` requiere que la declaración de la función sea

```
int encontrarMax(int x, int y)
```

Obsérvese que ésta es igual a la línea de encabezado de la función original para `encontrarMax()` con la sustitución de la palabra clave `int` en lugar de la palabra clave `void`.

[9]El tipo de datos devuelto sólo se relaciona con los tipos de datos de los parámetros en la medida en que el valor devuelto se calcula a partir de valores de parámetros. En este caso, en vista que la función se usa para devolver el valor máximo de sus parámetros, tendría poco sentido devolver un tipo de datos que no correspondiera con los tipos de parámetros de la función.

Habiendo declarado el tipo de datos que devolverá `encontrarMax()`, todo lo que resta es incluir una instrucción dentro de la función que cause la devolución del valor correcto. Para devolver un valor, una función debe usar una **instrucción de `devolución`**, la cual tiene la forma:[10]

return expresión;

Cuando se encuentra la instrucción de devolución, la expresión se evalúa primero. El valor de la expresión es convertido entonces de manera automática en el tipo de datos declarado en el encabezado de la función antes de ser enviado de regreso a la función que llama. Después que es devuelto el valor, el control del programa se revierte a la función que llama. Por tanto, para devolver el valor almacenado en `numMax`, todo lo que se necesita hacer es agregar la instrucción `return numMax;` antes de la llave de cierre de la función `encontrarMax()`. El código de la función completo es:

Éstos deberán ser del mismo tipo de datos

```
int encontrarMax(int x, int y) // linea de encabezado
                                  de la funcion
{                              // inicio del cuerpo de
                                  la funcion
  int numMax;                  // declaración de la variable

   if (x >= y)
     numMax = x;
   else
     numMax = y;

   return numMax;              // instruccion de devolucion
}
```

En este nuevo código para la función `encontrarMax()` hay que observar que el tipo de datos de la expresión contenida en la instrucción de devolución corresponde en forma correcta al tipo de datos en la línea de encabezado de la función. Le corresponde al programador asegurar que esto sea así en todas las funciones que devuelven un valor. Si hay una falla en hacer que corresponda con exactitud el valor devuelto con el tipo de datos declarado en la función, puede ser que no se produzca un error cuando su programa sea compilado, pero esto puede conducir a resultados indeseables debido a que el valor devuelto siempre es convertido al tipo de datos declarado en la declaración de la función. Por lo general esto es un problema sólo cuando la parte fraccionaria de un número de punto flotante o de precisión doble devuelta es truncada debido a que se declaró que la función devolvería un valor entero.

Habiendo cuidado el lado emisor de la transacción de devolución, ahora se debe preparar a la función que llama para recibir el valor enviado por la función llamada. En el lado que llama (receptor), la función que llama debe

- ser alertada del tipo de valor a esperar
- usar de manera apropiada el valor devuelto

[10]Muchos programadores colocan la expresión entre paréntesis, produciendo la instrucción `return (expresión);`. Aunque puede usarse cualquier forma, por consistencia sólo se adoptará una.

Para alertar a la función que llama acerca del tipo de valor devuelto que debe esperar se utiliza el prototipo de función. Por ejemplo, incluir el prototipo de función

```
int encontrarMax(int, int);
```

antes de la función `main()` es suficiente para alertar a `main()` que `encontrarMax()` es una función que devolverá un valor entero.

Para usar en verdad un valor devuelto hay que proporcionar una variable para almacenar el valor o usar el valor en forma directa en una expresión. Se logra almacenar el valor devuelto en una variable usando una instrucción de asignación estándar. Por ejemplo, puede usarse la instrucción de asignación

```
max = encontrarMax (primernum, segundonum);
```

para almacenar el valor devuelto por `encontrarMax()` en la variable llamada `max`. Esta instrucción de asignación hace dos cosas. Primera, el lado derecho de la instrucción de asignación llama a `encontrarMax()`, luego el resultado devuelto por `encontrarMax()` se almacena en la variable `max`. En vista que el valor devuelto por `encontrarMax()` es un número entero, la variable `max` deberá declararse también como una variable en número entero dentro de las declaraciones de variables de la función que llama.

El valor devuelto por una función no necesita almacenarse en forma directa en una variable, pero puede usarse siempre que una expresión sea válida. Por ejemplo la expresión `2 * encontrarMax(primernum, segundonum)` multiplica el valor devuelto por `encontrarMax()` por dos, y la instrucción

```
cout << encontrarMax(primernum, segundonum);
```

despliega el valor devuelto.

El programa 6.5 ilustra la inclusión de instrucciones de prototipo y de asignación para que `main()` declare, llame y almacene en forma correcta un valor devuelto por `encontrarMax()`. Como antes, y para conservar la convención de colocar primero la función `main()`, se ha colocado la función `encontrarMax()` después de `main()`.

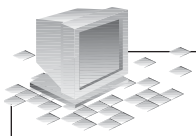
Al revisar el programa 6.5 es importante señalar los cuatro elementos que se han introducido en esta sección. El primer elemento es el prototipo para `encontrarMax()`. Esta instrucción, la cual termina con punto y coma, como todas las instrucciones de declaración, alerta a `main()` y a las funciones subsiguientes que usen `encontrarMax()` con el tipo de datos que devolverá `encontrarMax()`. El segundo elemento que se puede observar en `main()` es el uso de una instrucción de asignación para almacenar el valor devuelto de la llamada a `encontrarMax()` en la variable `numMax`. También se ha asegurado de declarar en forma correcta `numMax` como un número entero dentro de las declaraciones de variables de `main()` de modo que concuerde con el tipo de datos del valor devuelto.

Los últimos dos elementos que hay que observar conciernen a la codificación de la función `encontrarMax()`. La primera línea de `encontrarMax()` declara que la función devolverá un valor entero, y la expresión en la instrucción de devolución evalúa un tipo de datos que corresponda. Por tanto `encontrarMax()` es consistente internamente en enviar un valor entero de vuelta a `main()`, y `main()` ha sido alertado en forma correcta para recibir y usar el número entero devuelto.

Al escribir sus propias funciones debe tener en cuenta siempre estos cuatro elementos. Como ejemplo, vea si puede identificar estos cuatro elementos en el programa 6.6.

Programa 6.5

```
#include <iostream>
using namespace std;

int encontrarMax(int, int); // el prototipo de la función

int main()
{
  int primernum, segundonum, max;


  cout << "\nIntroduzca un numero: ";
  cin >> primernum;
  cout << "¡Estupendo! Por favor introduzca un segundo número: ";
  cin >> segundonum;

  max = encontrarMax(primernum, segundonum); // aquí se llama a la función

  cout << "\nEl máximo de los dos números es " << max << endl;

  return 0;
}

int encontrarMax(int x, int y)
{                     // inicio del cuerpo de la función
  int numMax;         // declaración de variable

  if (x >= y)         // encontrar el número máximo
    numMax = x;
  else
    numMax = y;

  return numMax;      // instrucción de devolución
}
```



Programa 6.6

```
#include <iostream>
using namespace std;

double convertir_temp(double); // prototipo de la función

int main()
{
  const CONVERSIONES = 4; // numero de conversiones que se harán
  int cuenta;
  double fahren;

  for(cuenta = 1; cuenta <= CONVERSIONES; cuenta++)
  {
    cout << "\nIntroduzca una temperatura en grados Fahrenheit: ";
    cin  >> fahren;
    cout << "El equivalente en grados Celsius es "
         << convertir_temp(fahren) << endl;
  }

  return 0;
}

// convertir Fahrenheit a Celsius
double convertir_temp(double in_temp)
{
  return (5.0/9.0) * (in_temp - 32.0);
}
```

Al revisar el programa 6.6, se analizará primero la función convertir_temp(). La definición completa de la función comienza con la línea de encabezado de la función y termina con la llave de cierre después de la instrucción de devolución. La función es declarada como `double`; esto significa que la expresión en la instrucción de devolución de la función debe evaluar a un número de precisión doble, lo cual hace. En vista que la línea de encabezado de la función no es una instrucción sino el inicio del código que define la función, la línea de encabezado de la función no termina con punto y coma.

En el lado receptor, `main()` tiene un prototipo para la función `convertir_temp()` que concuerda con la definición de la función de `convertir_temp()`. No se declara una variable en main() para almacenar el valor devuelto de `convertir_temp()` porque el valor devuelto es transmitido de inmediato a `cout` para su despliegue.

Vale la pena mencionar un punto más aquí. Uno de los propósitos de las declaraciones, como se aprendió en el capítulo 2, es alertar a la computadora de la cantidad de almacenamiento interno reservado para los datos. El prototipo para `convertir_temp()` ejecuta esta tarea y alerta al compilador del tipo de almacenamiento necesario para el valor devuelto. En vista que se ha elegido siempre enlistar `main()` como la primera función en un archivo, se deben incluir prototipos de función para todas las funciones llamadas por `main()` y cualesquiera funciones subsiguientes.

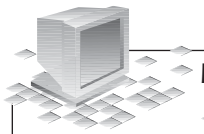
Funciones `inline`[11]

Llamar a una función pone una cierta cantidad de sobrecarga en la computadora, ya que consiste en colocar valores de argumentos en una región de memoria reservada a la que la función tiene acceso (esta región de memoria se conoce como **pila**), transmitir el control a la función, proporcionar una ubicación de memoria reservada para cualquier valor devuelto (de nuevo, la región de pila de la memoria se usa para este propósito) y por último regresar al punto apropiado en el programa que llama. Esta sobrecarga se justifica cuando una función es invocada muchas veces, porque puede reducir de manera significativa el tamaño de un programa. En lugar de repetir el mismo código cada vez que es necesario, el código se escribe una sola vez, como una función, y se llama cada vez que es necesario.

Sin embargo, para funciones pequeñas que no son llamadas muchas veces no se justifica la sobrecarga de transmitir y devolver valores. No obstante, aún sería conveniente agrupar las líneas repetitivas de código bajo un nombre de función común y hacer que el compilador coloque este código en forma directa en el programa siempre que se llame a la función. Esta capacidad es proporcionada por funciones `inline.`

Indicarle al compilador de C++ que una función es `inline` causa que una copia del código de la función sea colocada en el programa en el punto en que es invocada. la función. Por ejemplo, considérese la función `convertir_temp()` definida en el programa 6.6. En vista que ésta es una función relativamente corta, es ideal para ser una función `inline`. Para hacer a ésta, o a cualquier otra función, una función `inline`, tan sólo se requiere colocar la palabra reservada `inline` antes del nombre de la función y definir la función antes que se haga cualquier llamada. Esto se lleva a cabo para la función `convertir_temp()` en el programa 6.7.

[11]Esta sección es opcional y puede omitirse en la primera lectura sin perder la continuidad de la materia.



Programa 6.7

```
#include <iostream>
using namespace std;

inline double convertir_temp(double in_temp) // una función inline
{
  return (5.0/9.0) * (in_temp - 32.0);
}

int main()
{
  const CONVERSIONES = 4; // numero de conversiones que se harán
  int cuenta;
  double fahren;

  for(cuenta = 1; cuenta <= CONVERSIONES; cuenta++)
  {
    cout << "\nIntroduzca una temperatura en grados Fahrenheit: ";
    cin  >> fahren;
    cout << "El equivalente en grados Celsius es "
         << convertir_temp(fahren) << endl;
  }

  return 0;
}
```

En el programa 6.7 hay que observar que la función `inline` se coloca antes de cualesquier llamada a ella. Esto es un requisito de todas las funciones `inline` y evita la necesidad de un prototipo de función antes de cualquier función subsiguiente que llama. En vista que la función ahora es una `inline`, su código será expandido en forma directa siempre que sea llamada en el programa.

La ventaja de usar una función `inline` es que aumenta la velocidad de ejecución. En vista que la función `inline` es expandida en forma directa e incluida en cada expresión o instrucción que la llame, no se pierde tiempo de ejecución debido a la sobrecarga de la llamada y devolución requeridas por una función que no es `inline`. La desventaja es el aumento en el tamaño del programa cuando una función `inline` es llamada en forma repetida. Cada vez que se hace referencia a una función `inline`, se reproduce el código completo de la función y se almacena como una parte integral del programa. Una función que no es `inline`, sin embargo, se almacena en la memoria una sola vez. Sin importar cuántas veces sea llamada la función, se usa el mismo código. Por consiguiente, las funciones `inline` sólo deberán usarse para funciones pequeñas que no son llamadas en forma eshaustiva en un programa.

Ejercicios 6.2

1. Vuelva a escribir el programa 6.5 de modo que la función `encontrarMax()` acepte dos argumentos de precisión doble y devuelva un valor de precisión doble a `main()`. Asegúrese de modificar `main()` a fin de transmitir dos valores de punto flotante a `encontrarMax()` y aceptar y almacenar el valor de precisión doble devuelto por `encontrarMax()`.

2. Para los siguientes encabezados de función, determine el número, tipo y orden (secuencia) de los valores que deberán ser transmitidos a la función cuando sea llamada y el tipo de datos del valor devuelto por la función.

   a. `int factorial(int n)`
   b. `double voltios(int res, double induc, double cap)`
   c. `double potencia(int tipo, double induc, double cap)`
   d. `char marcador(char tipo, float corriente, float tiempo)`
   e. `int total(float cantidad, float tasa)`
   f. `float roi(int a, int b, char c, char d, float e, float f)`
   g. `void obtener_valor(int elemento, int iter, char decmarcador, char delim)`

3. Escriba encabezados de función para lo siguiente:

   a. una función llamada `verificar` que tenga tres parámetros. El primer parámetro deberá aceptar un número entero, el segundo un número de precisión doble y el tercer parámetro un número de precisión doble. La función no devuelve ningún valor.
   b. una función nombrada `encontrarAbs()` que acepte un número de precisión doble que se le transmite y devuelva su valor absoluto.
   c. una función nombrada `mult` que acepte dos números de punto flotante como parámetros, multiplique estos dos números y devuelva el resultado.
   d. una función nombrada `al_cuadrado()` que calcule y devuelva el cuadrado del valor entero que se le transmite.
   e. una función llamada `funpot()` que eleve un número entero que se le transmita a una potencia entera positiva (como un argumento) y devuelva el resultado como un número entero.
   f. una función que produzca una tabla de los números del 1 al 10, sus cuadrados y sus cubos. Que no se transmitan argumentos a la función y la función no devuelve ningún valor.

4. **a.** Escriba una función llamada `trianguloRectangulo()` que acepte el largo de dos lados de un triángulo rectángulo como los argumentos *a* y *b*, respectivamente. La subrutina deberá determinar y devolver la hipotenusa, *c*, del triángulo. (*Sugerencia*: Use el teorema de Pitágoras, $c^2 = a^2 + b^2$).
   b. Incluya la función escrita para el ejercicio 4a en un programa que funcione. La unidad de función `main()` deberá llamar en forma correcta a `trianguloRectangulo()` y desplegar el valor devuelto por la función.

5. a. Escriba una función en C++ llamada `encontrarAbs()` que acepte un número de precisión doble que se le transmite, calcule su valor absoluto y devuelva el valor absoluto a la función que llama. El valor absoluto de un número es el mismo número si éste es positivo y el negativo del número si éste es negativo.

b. Incluya la función escrita en el ejercicio 5a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y que devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

6. a. El volumen, v, de un cilindro está dado por la fórmula

$$v = \pi r^2 l$$

donde r es el radio del cilindro y l es su largo. Usando esta fórmula, escriba una función C++ nombrada `vol_cil()` que acepte el radio y el largo de un cilindro y devuelva su volumen.

b. Incluya la función escrita en el ejercicio 6a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

7. a. El área de la superficie, s, de un cilindro está dada por la fórmula

$$s = 2\pi r l$$

donde r es el radio del cilindro y l su largo. Usando esta fórmula escriba una función en C++ nombrada `areasup()` que acepte el radio y largo de un cilindro y devuelva su superficie.

b. Incluya la función escrita en el ejercicio 7a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelva en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

8. Un polinomio de segundo grado en x está dado por la expresión $ax^2 + bx + c$, donde a, b, y c son números conocidos y a no es igual a cero. Escriba una función en C++ nombrada *poli_dos(a, b, c, x)* que calcule y devuelva el valor de un polinomio de segundo grado para valores transmitidos de *a, b, c* y *x*.

9. a. La deflexión máxima permisible de una viga depende de su función. Para un piso, la deflexión máxima permisible típica, en pulgadas, es $D_{máx} = L / 240$, mientras para una viga del techo $D_{máx} = L / 180$, donde L es el largo de la viga en pulgadas. Usando estas fórmulas, escriba y pruebe una función nombrada `deflexMax()` que acepte el largo de una viga, en pies, y el tipo de viga (piso o techo) como un código de carácter y devuelva la deflexión máxima permisible.

b. Incluya la función escrita en el ejercicio 9a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

10. a. La carga, P_{cr}, en unidades de kilolibras (klb), aplicada a una columna que causará que la columna se combe se conoce como la carga de combamiento crítica. Esta carga puede determinarse usando la ecuación

$$P_{cr} = \pi^2 E A / (L / r)^2$$

donde E es el módulo de elasticidad del material usado en la columna, A es el área de corte transversal, L es el largo de la columna y r es su radio de giro. Usando esta fórmula, escriba una función en C++ llamada `cargaC()` que acepte valores de E, A, L y r, y devuelva la carga crítica.

b. Incluya la función escrita en el ejercicio 10a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

11. a. Un algoritmo de programación útil en extremo para redondear un número real a *n* lugares decimales es

Paso 1: Multiplicar el número por 10^n

Paso 2: Sumar 0.5

Paso 3: Eliminar la parte fraccionaria del resultado

Paso 4: Dividir entre 10^n

Por ejemplo, usar este algoritmo para redondear el número 78.374625 a tres lugares decimales produce:

Paso 1: $78.374625 \times 10^3 = 78374.625$

Paso 2: $78374.625 + 0.5 = 78375.125$

Paso 3: Conservar la parte entera = 78375

Paso 4: 78375 dividido entre $10^3 = 78.375$

Usando este algoritmo, escriba una función en C++ que acepte un valor introducido por un usuario y devuelva el resultado redondeado a dos lugares decimales.

b. Introduzca, compile y ejecute el programa escrito para el ejercicio 11a.

12. a. Escriba una función en C++ llamada `entero()` que devuelva la parte entera de cualquier número que se transmita a la función. (*Sugerencia:* Asigne el argumento transmitido a una variable entera.)

b. Incluya la función escrita en el ejercicio 12a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

13. a. Escriba una función en C++ nombrada `partefrac()` que devuelva la parte fraccionaria de cualquier número transmitido a la función. Por ejemplo, si se transmite el número 256.879 a `partefrac()`, debería devolverse el número 0.879. Haga que la función `partefrac()` llame a la función `entero()` que escribió en el ejercicio 12. El número devuelto puede determinarse entonces como el número transmitido a `partefrac()` menos el valor devuelto cuando el

mismo argumento es transmitido a `entero()`. El programa completo deberá consistir de `main()` seguido por `partefrac()` seguido por `entero()`.

b. Incluya la función escrita en el ejercicio 13a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.

14. Todos los años que son divisibles entre 400 o son divisibles entre cuatro y no son divisibles entre 100 son años bisiestos. Por ejemplo, en vista que 1600 es divisible entre 400, el año 1600 fue un año bisiesto. Del mismo modo, en vista que 1988 es divisible entre cuatro pero no entre 100, el año 1988 también fue un año bisiesto. Usando esta información, escriba una función en C++ que acepte el año como entrada de un usuario y devuelva un uno si el año transmitido es un año bisiesto o un cero si no lo es.

6.3 DEVOLVER VALORES MÚLTIPLES

En una invocación de una función típica, la función llamada recibe valores de la función que la llama, almacena y manipula los valores transmitidos y devuelve en forma directa cuando mucho un valor único. Cuando los datos son transmitidos en esta manera se conoce como **transmisión por valor**.

Llamar a una función y transmitir argumentos por valor es una ventaja distintiva de C++. Permite que se escriban funciones como entidades independientes que pueden usar cualquier nombre de variable o parámetro sin preocuparse por que otras funciones puedan usar también el mismo nombre. También alivia cualquier preocupación de que alterar un parámetro o variable en una función pueda alterar de manera inadvertida el valor de una variable en otra función. Bajo este enfoque, los parámetros pueden considerarse como variables inicializadas o variables a las que se les asignarán valores cuando se ejecute la función. Sin embargo, en ningún momento la función llamada tiene acceso directo a cualquier variable definida en la función que llama, aun si la variable se usa como un argumento en la llamada a la función.

Hay ocasiones, sin embargo, en que es necesario alterar este enfoque dándole a una función llamada acceso directo a las variables de su función que llama. Esto le permite a la función, la cual es la función llamada, usen y cambien el valor de variables que se han definido en la función que llama. Para hacer esto se requiere que la dirección de la variable se transmita a la función llamada. Una vez que la función llamada tiene la dirección de la variable, "sabe dónde vive la variable", por así decirlo, y puede tener acceso y cambiar el valor almacenado ahí en forma directa.

La transmisión de direcciones se conoce como función de **transmisión por referencia,**[12] en vista que la función llamada puede hacer referencia, o tener acceso, a la variable cuya dirección se ha transmitido. C++ proporciona dos tipos de parámetros de dirección, referencias y apuntadores. En esta sección se describe el método que usa parámetros de referencia.

[12]También se conoce como **llamada por referencia,** donde, una vez más, el término se aplica sólo a los argumentos cuya dirección se ha transmitido.

Transmisión y uso de parámetros de referencia

Como siempre, al intercambiar datos entre dos funciones se deben atender los lados emisor y receptor del intercambio de datos. Desde el lado emisor, sin embargo, llamar a una función y transmitir una dirección como un argumento que se aceptará como parámetro de referencia es exactamente lo mismo que llamar a una función y transmitir un valor; la función llamada es convocada a la acción proporcionando su nombre y una lista de argumentos. Por ejemplo, la instrucción `valnuevo(primernum, segundonum);` llama a la función nombrada `valnuevo` y le transmite dos argumentos. Que se transmita en realidad un valor o una dirección depende de los tipos de parámetros declarados para `valnuevo()`. Ahora se escribirá la función y el prototipo `valnuevo` de modo que reciban las direcciones de las variables `primernum` y `segundonum`, las cuales se supondrá que son variables de precisión doble, en lugar de sus valores.

Uno de los primeros requisitos al escribir `valnuevo()` es declarar dos parámetros de referencia para aceptar direcciones transmitidas. En C++ se declara un parámetro de referencia usando la sintaxis

tipo-de-datos& nombre-de-referencia

Por ejemplo, la declaración de referencia

```
double& num1;
```

declara que num1 es un parámetro de referencia que se utilizará para almacenar la dirección de un `double`. Del mismo modo, `int& segundonum` declara que `segundonum` es una referencia a un número entero y `char& clave` declara que `clave` es una referencia a un carácter.

Se recordará de la sección 2.4 que el símbolo ampersand, &, en C++ significa "la dirección de". Además, cuando se usa un símbolo & dentro de una declaración se refiere a "la dirección del" tipo de datos precedente. Usando esta información, declaraciones como `double& num1` e `int& segundonum` a veces se entienden con más claridad si se leen al revés. Leer la declaración `double& num1` de esta manera produce la información que "num1 es la dirección de un valor de precisión doble".

En vista de que es necesario aceptar dos direcciones en la lista de parámetros para `valnuevo()`, pueden usarse las declaraciones `double& num1, double& num2`. Al incluir estas declaraciones dentro de la lista de parámetros para `valnuevo()`, y suponer que la función no devuelve ningún valor `(void)`, el encabezado de función para `valnuevo()` se vuelve:

```
void valnuevo(double& num1, double& num2)
```

Para la línea de encabezado de esta función, un prototipo de función apropiado es

```
void valnuevo(double&, double&);
```

Este prototipo y línea de encabezado se han incluido en el programa 6.8, el cual incorpora un cuerpo de función `valnuevo()` completo que despliega y altera en forma directa los valores almacenados en estas variables de referencia desde dentro de la función que se ha llamado.

Programa 6.8

```
#include <iostream>
using namespace std;

void valnuevo(double&, double&); // prototipo con dos parámetros de referencia

int main()
{
  double primernum, segundonum;

  cout << "Introduzca dos números: ";
  cin >> primernum >> segundonum;
  cout << "\nEl valor en primernum es: " << primernum << endl;
  cout << "El valor en segundonum es: " << segundonum << "\n\n";

  valnuevo(primernum, segundonum); // llamada a la función

  cout << "Ahora el valor en primernum es: " << primernum << endl;
  cout << "Ahora el valor en segundonum es: " << segundonum << endl;

  return 0;
}

void valnuevo(double& xnum, double& ynum)
{
  ccout << "El valor en xnum es: " << xnum << endl;
  cout << "El valor en ynum es: " << ynum << "\n\n";
  xnum = 89.5;
  ynum = 99.5;

  return;
}
```

Al llamar a la función `valnuevo()` dentro del programa 6.8, es importante entender la conexión entre los argumentos, `primernum` y `segundonum`, usados en la llamada a la función y los parámetros, `xnum` y `ynum`, usados en el encabezado de la función. *Ambos se refieren a los mismos elementos de datos.* La importancia de esto es que los valores en los argumentos (`primernum` y `segundonum`) ahora pueden alterarse desde dentro de `valnuevo()` usando los nombres de parámetros (`xnum y ynum`). Por tanto, el parámetro `xnum` y `ynum` no almacena copias de los valores en `primernum` y `segundonum`, sino que tiene acceso directo a las ubicaciones en la memoria apartados para estos dos argumentos. La equivalencia de los nombres de argumentos en el programa 6.8, la cual es la esencia de una transmisión por referencia, se ilustra en la figura 6.8. Como se muestra en esta figura, los nombres de argumentos y sus nombres de parámetros correspondientes tan sólo son nombres diferentes que se refieren a las mismas áreas de almacenamiento en la memoria. En `main()` se hace referencia a estas ubicaciones de memoria por los nombres

primernum y segundonum, respectivamente, mientras en valnuevo() se hace referencia a las mismas ubicaciones con los nombres de parámetro xnum y ynum, respectivamente.

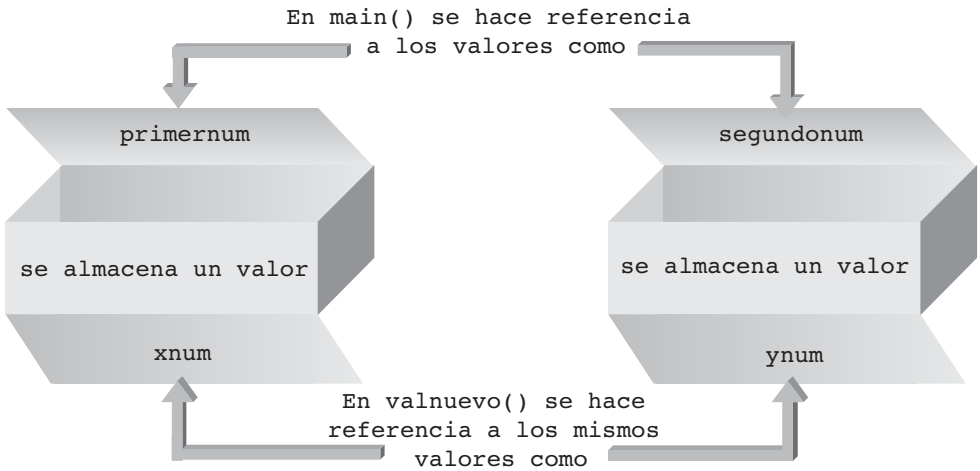


Figura 6.8 *La equivalencia de argumentos y parámetros en el programa 6.8.*

El siguiente ejemplo muestra la ejecución que se obtuvo usando el programa 6.8:

```
Introduzca dos números: 22.5 33.0

El valor en primernum es: 22.5
El valor en segundonum es:   33

El valor en xnum es: 22.5
El valor en ynum es: 33

Ahora el valor en primernum es: 89.5
Ahora el valor en segundonum es: 99.5
```

Al revisar esta salida, hay que observar que los valores desplegados al principio para los parámetros xnum y ynum son los mismos que los desplegados para los argumentos primernum y segundonum. Sin embargo, en vista que xnum y ynum son parámetros de referencia, valnuevo() ahora tienen acceso directo a los argumentos primernum y segundonum. Por tanto, cualquier cambio a xnum dentro de valnuevo() altera en forma directa el valor de primernum en main() y cualquier cambio a ynum cambia en forma directa el valor de segundonum. Como lo ilustran los valores finales desplegados, la asignación de valores a xnum y ynum dentro de valnuevo() se refleja en main() como la alteración de los valores de primernum y segundonum.

La equivalencia entre los argumentos reales que llaman y los parámetros de función ilustrados en el programa 6.8 proporcionan la base para devolver valores múltiples desde adentro de una función. Por ejemplo, suponga que se requiere una función para aceptar tres valores, calcular la suma y el producto de estos valores y devolver estos resultados calculados a la rutina que llama. Al nombrar a la función calcular() y proporcionar cinco parámetros (tres para los datos de entrada y dos referencias para los valores devueltos), puede usarse la siguiente función.

```
void calcular(double num1, double num2, double num3, double& total, double& producto)
{
  total = num1 + num2 + num3;
  producto = num1 * num2 * num3;
  return;
}
```

Esta función tiene cinco parámetros, nombrados `num1`, `num2`, `num3`, `total` y `producto`, de los cuales sólo los últimos dos son declarados como referencias. Por tanto, los primeros tres argumentos son transmitidos por valor y los últimos dos argumentos son transmitidos por referencia. Dentro de la función sólo se alteran los dos últimos parámetros. El valor del cuarto parámetro, `total`, se calcula como la suma de los primeros tres parámetros y el último parámetro, `producto`, se calcula como el producto de los parámetros `num1`, `num2` y `num3`. El programa 6.9 incluye esta función en un programa completo.

Programa 6.9

```
#include <iostream>
using namespace std;

void calcular(double, double, double, double&, double&); // prototipo

int main()
{
  double primernum, segundonum, tercernum, suma, producto;

  cout << "Introduzca tres números: ";
  cin >> primernum >> segundonum >> tercernum;

  calcular(primernum, segundonum, tercernum, suma, producto); // llamada a la función

  cout << "\nLa suma de los números es: " << suma << endl;
  cout << "El producto de los números es: " << producto << endl;

  return 0;
}

void calcular(double num1, double num2, double num3, double& total, double& producto)
{
  total = num1 + num2 + num3;
  producto = num1 * num2 * num3;
  return;
}
```

Dentro de `main()`, la función `calcular()` es llamada usando los cinco argumentos `primernum`, `segundonum`, `tercernum`, suma y producto. Como se requiere, estos argumentos concuerdan en número y tipo de datos con los parámetros declarados por `calcular()`. De los cinco argumentos transmitidos, sólo a `primernum`, `segundonum` y `tercernum` se les han asignado valores cuando se hace la llamada a `calcular()`. Los dos argumentos restantes no se han inicializado y se usarán para recibir valores devueltos por `calcular()`. Dependiendo del compilador utilizado para compilar el programa, estos argumentos contendrán al principio ceros o valores "basura". La figura 6.9 ilustra la relación entre los nombres reales y de parámetro y los valores que contienen después de ser devueltos de `calcular()`.

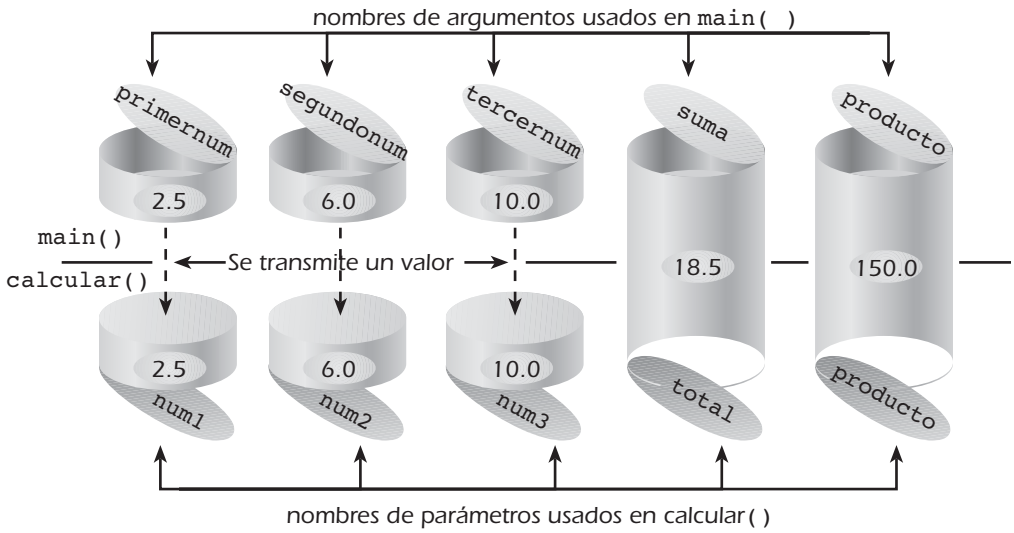


Figura 6.9 *Relación entre los nombres de argumento y de parámetro.*

Una vez que es llamada `calcular()`, usa sus primeros tres parámetros para calcular valores para total y producto y luego devuelve el control a `main()`. Debido al orden de los argumentos reales de la función que llama, `main()` conoce los valores calculados por `calcular()` como suma y producto, los cuales son desplegados luego. A continuación se presenta una muestra de ejecución usando el programa 6.9.

```
Introduzca tres números: 2.5 6.0 10.0
La suma de los números introducidos es: 18.5
El producto de los números introducidos es: 150
```

Como un ejemplo final que ilustra la utilidad de transmitir referencias a una función llamada, se construirá una función llamada `intercambio()` que intercambia los valores de dos de las variables de precisión doble de `main()`. Una función así es útil cuando se clasifica una lista de números.

En vista que es afectado el valor de más de una variable, `intercambio()` no puede escribirse como una función de transmisión por valor que devuelve un valor único. El intercambio deseado de las variables de `main()` por `intercambio()` sólo puede obtenerse dándole a `intercambio()` acceso a las variables de `main()`. Una forma de hacer esto es usar parámetros de referencia.

Ya se ha visto cómo transmitir referencias a dos variables en el programa 6.8. Ahora se construirá una función para intercambiar los valores en los argumentos de referencia transmitidos. Intercambiar valores en dos variables se logra usando el algoritmo de intercambio en tres pasos:

1. Guardar el valor del primer parámetro en una ubicación temporal (véase la figura 6.10a).
2. Almacenar el valor del segundo parámetro en la primera variable (véase la figura 6.10b).
3. Almacenar el valor temporal en el segundo parámetro (véase la figura 6.10c).

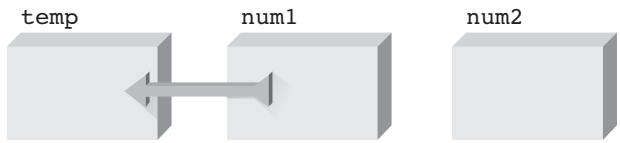


Figura 6.10a *Guardar el primer valor.*

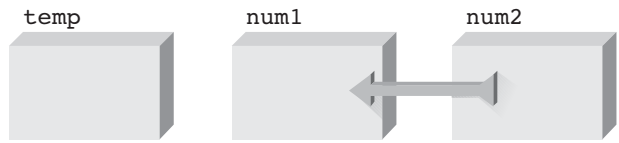


Figura 6.10b *Reemplazar el primer valor con el segundo valor.*

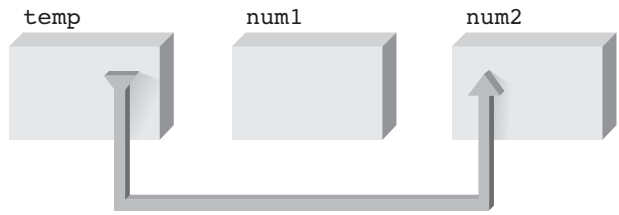


Figura 6.10c *Cambiar el segundo valor.*

A continuación está la función `intercambio()` escrita de acuerdo con estas especificaciones:

```
void intercambio(double& num1, double& num2)
{
  double temp;

  temp = num1;      // guarda el valor de num1
  num1 = num2;      // almacena el valor de num2 en num1
  num2 = temp;      // cambia el valor de num2

  return;
}
```

Hay que observar que el uso de referencias en la línea de encabezado de `intercambio()` le da acceso a `intercambio()` a los argumentos equivalentes en la función que llama. Por

tanto, cualquier cambio a los dos parámetros de referencia en `intercambio()` de manera automática cambia los valores en los argumentos de la función que llama. El programa 6.10 contiene `intercambio()` en un programa completo.

Programa 6.10

```
#include <iostream>
using namespace std;

void intercambio(double&, double&); // la función recibe dos referencias

int main()
{
  double primernum = 20.5, segundonum = 6.25;

  cout << "El valor almacenado en primernum es: " << primernum << endl;
  cout << "El valor almacenado en segundonum es : " << segundonum << "\n\n";

  intercambio(primernum, segundonum); // llama a la función con referencias

  cout << "Ahora el valor almacenado en primernum es: "
       << primernum << endl;
  cout << "Ahora el valor almacenado en segundonum es: "
       << segundonum << endl;

  return 0;
}

void intercambio(double& num1, double& num2)
{
  double temp;

  temp = num1;      // guarda el valor de num1
  num1 = num2;      // almacena el valor de num2 en num1
  num2 = temp;      // cambia el valor de num2

  return;
}
```

La siguiente muestra de ejecución se obtuvo usando el programa 6.10:

```
El valor almacenado en primernum es: 20.5
El valor almacenado en segundonum es: 6.25


Ahora el valor almacenado en primernum es: 6.25
Ahora el valor almacenado en segundonum es:  20.5
```

Como lo ilustra su salida, los valores almacenados en las variables de `main()` se han modificado desde dentro de `intercambio()`, lo cual fue posible por el uso de parámetros de referencia. Si se hubiera utilizado una transmisión por valor, el intercambio dentro de `intercambio()` sólo habría afectado los parámetros de `intercambio()` y no lograría nada con respecto a las variables de `main()`. Por tanto, una función como `intercambio()` sólo puede escribirse usando referencias o algún otro medio que proporcione acceso a las variables de `main()` (este otro medio se llava a cabo por apuntadores, tema del capítulo 12).

Al usar argumentos de referencia, es necesario mencionar dos precauciones. La primera es que los argumentos de referencia deben ser variables (es decir, no pueden usarse para cambiar constantes). Por ejemplo, llamar a `intercambio()` con dos constantes, como en la llamada `intercambio(20.5, 6.5)` transmite dos constantes a la función. Aunque `intercambio()` puede ejecutarse, no cambiará los valores de estas constantes.[13]

La segunda precaución es que una llamada a función en sí no da indicio de que la función llamada usará parámetros de referencia. La opción por omisión en C++ es hacer transmisiones por valor en lugar de transmisiones por referencia, precisamente para limitar la capacidad de una función llamada para alterar variables en la función que llama. Este procedimiento de llamada deberá acatarse siempre que sea posible, lo cual significa que los parámetros de referencia sólo deberían usarse en situaciones muy restringidas que en realidad requieran múltiples valores devueltos, como en la función `intercambio()` ilustrada en el programa 6.10. La función `calcular()`, incluida en el programa 6.9, aunque útil con propósitos ilustrativos, también podría ser escrita como dos funciones separadas, cada una devolviendo un valor único.

Ejercicios 6.3

1. Escriba declaraciones de parámetros para lo siguiente:
   a. un parámetro nombrado cantidad que será una referencia a un valor de precisión doble.
   b. un parámetro nombrado precio que será una referencia a un número de precisión doble.
   c. un parámetro nombrado minutos que será una referencia a un número entero.
   d. un parámetro nombrado `clave` que será una referencia a un carácter.
   e. un parámetro nombrado `rendimiento` que será una referencia a un número de precisión doble.

2. Se van a usar tres argumentos en número entero en una llamada a una función denominada `tiempo()`. Escriba un encabezado de función adecuado para `tiempo()`, suponiendo que `tiempo()` acepte estas variables como los parámetros de referencia `seg`, `min` y `horas`, y no devuelva ningún valor a la función que llama.

3. Vuelva a escribir la función `encontrarMax()` del programa 6.5 de modo que la variable `max`, declarada en `main()`, se use para almacenar el valor máximo de los dos números transmitidos. El valor de `max` deberá establecerse en forma directa desde dentro de `encontrarMax()`. (*Sugerencia:* `encontrarMax()` tendrá que aceptar una referencia a `max`.)

[13] Muchos compiladores tomarán esto como un error.

4. Escriba una función nombrada `cambio()` que tenga un parámetro en número entero y seis parámetros de referencia en número entero nombrados cien, `cincuenta`, `veinte`, `diez`, `cinco y uno`, respectivamente. La función tiene que considerar el valor entero transmitido como una cantidad en dólares y convertir el valor en el número menor de billetes equivalentes. Usando las referencias, la función deberá alterar de manera directa los argumentos respectivos en la función que llama.

5. Escriba una función nombrada `tiempo()` que tenga un parámetro en número entero llamado segundos y tres parámetros de referencia enteros nombrados `horas`, `min` y `seg`. La función es convertir el número de segundos transmitido en un número equivalente de horas, minutos y segundos. Usando las referencias la función deberá alterar de manera directa los argumentos respectivos en la función que llama.

6. Escriba una función nombrada `calc_años()` que tenga un parámetro entero que represente el número total de días desde la fecha 1/1/1900 y parámetros de referencia nombrados `año`, `mes` y `día.` La función es calcular el año, mes y día actual para el número dado de días que se le transmitan. Usando las referencias, la función deberá alterar en forma directa los argumentos respectivos en la función que llama. Para este problema suponga que cada año tiene 365 días y cada mes tiene 30 días.

7. El siguiente programa usa los mismos nombres de argumento y parámetro tanto en la función que llama como en la función llamada. Determine si esto causa algún problema para la computadora.

```
#include <iostream>
using namespace std;

void tiempo(int&, int&); // prototipo de la función

int main()
{
  int min, hora;

  cout << "Introduzca dos números :";
  cin >> min >> hora;
  tiempo(min, hora);

  return 0;
}

void tiempo(int& min, int& hora) // acepta dos referencias
{
  int seg;

  seg = (hora * 60 + min) * 60;
  ccout << "El número total de segundos es " << seg <<
  endl;

  return;
}
```

6.4 APLICACIONES

Preparar un programa de computadora bien diseñado es muy parecido a preparar un reporte semestral bien diseñado; ambos deben comenzar con un esquema, el cual puede ser escrito o, para programas muy pequeños, puede tan sólo mantenerse en la mente conforme se desarrolla el programa. Sin embargo, como sucede con un esquema para un reporte semestral, el cual enumera los temas principales del reporte, el esquema inicial de un programa de computadora debe proporcionar un listado de las tareas primarias que debe cumplir el programa.

En forma escrita, el esquema inicial de un programa de computadora por lo general es una descripción en seudocódigo (véase la sección 1.3) o un diagrama de estructura de primer nivel (véase la sección 1.2). Este esquema inicial comienza el proceso de definir un problema más complicado en un conjunto de tareas más pequeñas y más manejables. Cada una de estas tareas puede subdividirse o depurarse más, en tareas aún más pequeñas, si se requiere. Una vez que las tareas están bien definidas, el trabajo real de codificación puede comenzar, iniciando con cualquier tarea, en cualquier orden. Si hay más tareas de las que puedan ser manejadas por un programador, pueden distribuirse entre tantos programadores como se requiera. Esto es equivalente a tener a muchas personas trabajando en un proyecto de investigación grande, con cada persona responsable de un tema individual. Un esquema general aplicable a muchas tareas de ingeniería y científicas es el siguiente algoritmo:

Obtener las entradas del problema
Calcular el resultado deseado
Reportar los resultados del cálculo

Estas tres tareas son las responsabilidades primarias de todo programa, y se hará referencia a este algoritmo como el **algoritmo para resolver problemas.** Un diagrama de estructura de primer nivel de este algoritmo se muestra en la figura 6.11.

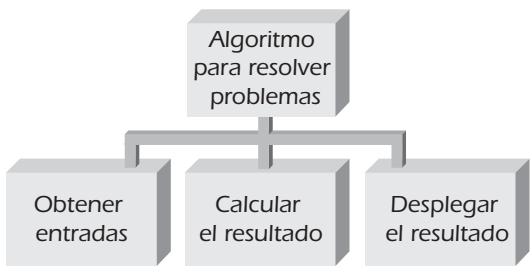


Figura 6.11 *Diagrama de estructura de primer nivel del algoritmo para resolver problemas.*

Cada tarea en el algoritmo para resolver problemas puede trabajarse en forma independiente como una función, una especie de "mini" programa en C++ que de manera típica es más fácil de completar que un programa entero. Cada una de estas tareas de función puede refinarse y codificarse en cualquier orden deseado, aunque completar primero la sección de entrada por lo general hace más fácil la prueba y el desarrollo. Ahora aplicaremos este procedimiento de desarrollo a un problema de programación real.

Aplicación 1: Conversión de coordenadas rectangulares a polares

Suponga que debemos escribir un programa en C++ para convertir las coordenadas rectangulares (*x*,*y*) de un punto en forma polar. Es decir, dada una posición *x* y *y* en un sistema de coordenadas cartesiano, como se ilustra en la figura 6.12, se debe calcular la distancia desde el origen, *r*, y el ángulo desde el eje *x*, θ, especificado por el punto. Los valores de *r* y θ se conocen como las *coordenadas polares* del punto.

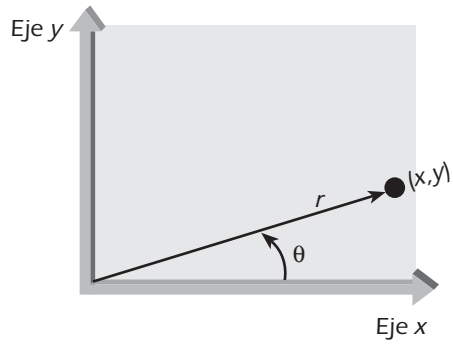


Figura 6.12 Correspondencia entre coordenadas polares (distancia y ángulo) y cartesianas (x,y).

Cuando se conocen las coordenadas *x* y *y* de un punto, las coordenadas *r* y θ equivalentes pueden calcularse usando las fórmulas:

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \tan^{-1}(y / x) \quad x \neq 0$$

Se inicia el desarrollo del programa con un esquema de lo que éste ha de lograr. Puede construirse una descripción primaria en seudocódigo del programa deseado usando el algoritmo para resolver problemas en lo que se refiere a los detalles de esta aplicación. Las entradas requeridas son una coordenada *x* y una coordenada *y*, el cálculo es convertir los valores introducidos en su forma de coordenadas polares, y el despliegue son las coordenadas polares calculadas. Por tanto, la descripción de seudocódigo inicial es

Obtener los valores de las coordenadas x y y
Calcular los valores de las coordenadas polares (r y θ)
Desplegar los valores de las coordenadas polares

El diagrama de estructura de primer nivel o nivel superior equivalente para este algoritmo se ilustra en la figura 6.13.

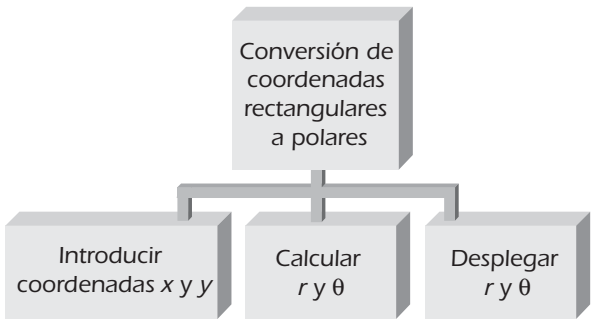


Figura 6.13 *Diagrama de estructura de nivel superior.*

Como éste es un programa relativamente simple y cada tarea descrita por el algoritmo está bien definida, se puede iniciar la codificación de cada tarea. Para ilustrar que cada tarea puede ser codificada de manera independiente de cualquier otra tarea, se comenzará de manera arbitraria con la codificación de la función que realiza el cálculo de las coordenadas polares. Como una característica adicional, esta función devolverá el ángulo θ en grados en lugar de la medida radianes devuelta por la función intrínseca `atan()`. En vista que esta función debe recibir dos entradas, las coordenadas *x* y *y*, y devolver dos salidas, las coordenadas *r* y θ, se le proporcionan a la función cuatro parámetros, dos para sus entradas y dos para sus salidas. Seleccionando de manera arbitraria los nombres de parámetros de `x`, `y`, `r`, y `theta`, y nombrando a la función `polar()`, el siguiente código lleva a cabo el cálculo requerido de coordenadas polares.

```
void polar(double x, double y, double& r, double& theta)
{
  const double A_GRADOS = 180.0/3.141593;

  r = sqrt(x * x + y * y);
  theta = atan(y/x) * A_GRADOS;

  return;
}
```

La función `polar()` es bastante sencilla. La línea de encabezado de la función declara que la función no devolverá en forma directa ningún valor y cada uno de sus parámetros es declarado como un tipo de datos en precisión doble. Los primeros dos parámetros se usarán para aceptar un valor *x* y uno *y*, mientras los últimos dos parámetros, los cuales son parámetros de referencia, se usarán para transmitir los valores de distancia y ángulo convertidos de vuelta a la función que llama. Dentro del cuerpo de la función se define una constante llamada `A_GRADOS` como el factor 180.0/3.142593. Las siguientes dos instrucciones de asignación usan los dos parámetros, `x` y `y`, para asignar valores a los parámetros *`r`* y `theta`. La constante nombrada `A_GRADOS` se usa para convertir el valor en radianes devuelto de la función `atan()` a grados. Como está escrita, la función `polar()` puede ser complicada para verificar cualquier error en tiempo de compilación.

Para entender cómo se transmiten los valores devueltos es útil pensar en los parámetros de referencia `r` y `theta` como recipientes (o variables) a través de los cuales pueden transmitirse valores en cualquier dirección. Esta situación se muestra en la figura 6.14, la

cual es útil para entender las características fundamentales de los parámetros de referencia: *tan sólo proporcionan la capacidad para que una función llamada y una que llama tengan acceso a la misma área de almacenamiento usando nombres diferentes.*

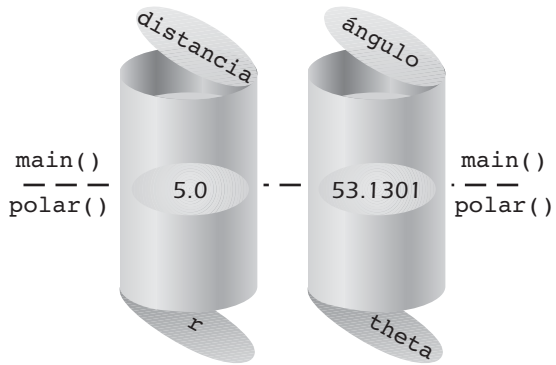


Figura 6.14 *Valores de los parámetros cuando se llama a* `polar()`.

Como se muestra en la figura 6.14, la función que llama puede tener acceso a los valores asignados a `r` y `theta` dentro de `polar()` usando los nombres de argumento `distancia` y `angulo`, o cualesquier otros nombres de argumentos seleccionados por el programador.

Probar la función

Una vez que se ha escrito `polar()`, puede probarse en forma independiente de cualquier otra función. Esto se hace escribiendo una función controladora dedicada que sólo llame a `polar()`, como se hace en el programa 6.11.

Programa 6.11

```
#include <iostream>
#include <cmath>
using namespace std;

void polar(double, double, double&, double&); // prototipo de la función

int main()
{
  double distancia, angulo;
  polar(3.0, 4.0, distancia, angulo);
  cout << "r = " << distancia << endl;
  cout << "ángulo = " << angulo << endl;

  return 0;
}
```

(Continúa)

(Continuación)

```
void polar(double x, double y, double& r, double& theta)
{
  const double A_GRADOS = 180.0/3.141593;

  r = sqrt(x * x + y * y);
  theta = atan(y/x) * A_GRADOS;

  return;
}
```

Hay que observar que en `main()` se transmitieron las constantes 3.0 y 4.0 a `polar()`. La función acepta estas entradas como los parámetros `x` y `y`, y usa estos parámetros para calcular valores para los parámetros r y theta. Dentro de `main()`, estos dos últimos parámetros se conocen como distancia y angulo, cuyos valores se despliegan inmediatamente después que se hace la llamada a `polar()`. La salida producida cuando se ejecuta el programa 6.11 es

```
r = 5
angulo = 53.1301
```

Éstos son los mismos resultados que se obtendrían por medio de un cálculo manual. Cómo la función lleva a cabo sólo dos cálculos, y el resultado desplegado por el programa de prueba concuerda por medio del obtenido con el cálculo manual, la función ha sido probada por completo por sí misma. Aún resta probarla en grupo con las dos funciones restantes requeridas por el programa completo para asegurar que los valores de argumentos correctos se intercambian entre cada función.

Completar el Programa

El diagrama de estructura para el programa completo (figura 6.13) requiere que también se escriban funciones para aceptar dos coordenadas rectangulares y desplegar las coordenadas polares calculadas, respectivamente. La siguiente función, `obtrec()`, puede utilizarse para aceptar los datos de entrada.

```
void obtrec(double& x, double& y)
{
  cout << "Programa de conversión de coordenadas"
       << " rectangulares a polares\n" << endl;
  cout << "Introduzca la coordenada x: ";
  cin  >> x;
  cout << "Introduzca la coordenada y: ";
  cin  >> y;

  return;
}
```

En esta función se usan los parámetros de referencia `x` y `y` para devolver los valores que se introducen en respuesta a los dos indicadores `cin`. Como con la función `polar()`, esta función puede probarse por sí misma usando un pequeño programa controlador dedicado. La función con su programa controlador se ilustra en el programa 6.12.

Programa 6.12

```
#include <iostream>
using namespace std;

void obtrec(double&, double&); // prototipo de la función

int main()
{
  double xcoord, ycoord;

  obtrec(xcoord, ycoord);
  cout << "El valor introducido para x es " << xcoord << endl;
  cout << "El valor introducido para y es " << ycoord << endl;

  return 0;
}


void obtrec(double& x, double& y)
{
  cout << "Programa de conversión de coordenadas"
       << " rectangulares a polares\n" << endl;
  cout << "Introduzca la coordenada x: ";
  cin  >> x;
  cout << "Introduzca la coordenada y: ";
  cin  >> y;

  return;
}
```

Observe que el programa controlador dedicado, el cual también se conoce como controlador de "aplicación para el usuario" o "de interfaz", se ha usado tanto para llamar a `obtrec()` como para desplegar los valores devueltos por esta función. La siguiente salida producida por el programa 6.12 verifica la operación correcta de la función `obtrec()`:

```
Programa de conversión de coordenadas rectangulares a
polares

Introduzca la coordenada x: 3
Introduzca la coordenada y: 4
```

```
El valor introducido para x es 3
El valor introducido para y es 4
```

De una manera similar, se construye la función para desplegar las coordenadas polares. El programa 6.13 contiene tanto la función, la cual se ha nombrado `mostrarlo()`, como un controlador de interfaz usado para probar la función. Hay que observar que los nombres de parámetros usados en la línea de encabezado para `mostrarlo()` no necesitan ser iguales que los usados en cualquier otra función. `mostrarlo()` se construye tan sólo para desplegar los valores en sus dos parámetros, los cuales en este caso se han nombrado `radio` y `angulo`.

Programa 6.13

```
#include <iostream>
using namespace std;

void mostrarlo(double, double); // prototipo de la función

int main()
{
  mostrarlo(5.0, 53.1301);

  return 0;
}

void mostrarlo(double radio, double angulo)
{
  cout << "\nLas coordenadas polares son: " << endl;
  cout << " Distancia desde el origen: " << radio << endl;
  cout << " Angulo (en grados) respecto al eje x: " << angulo << endl;

  return;
}
```

La salida del programa 6.13, la cual se muestra a continuación, verifica que `mostrarlo()` despliega en forma correcta los valores que se le transmiten.

```
Las coordenadas polares son:
  Distancia desde el origen: 5
  Angulo (en grados) respecto al eje x: 53.1301
```

Ahora falta crear un programa `main()` que llame a cada una de las funciones desarrolladas en el orden correcto. Esto se hace en el programa 6.14, el cual también incluye las funciones `obtrec()`, `polar()` y `mostrarlo()`.

Programa 6.14

```
// Este programa convierte coordenadas rectangulares en coordenadas polares
// Funciones usadas: obtrec() - obtener las coordenadas rectangulares
//                 : polar() - calcular las coordenadas polares
//                 : mostrarlo() - desplegar las coordenadas polares
//
#include <iostream>
#include <cmath>
using namespace std;

void obtrec(double&, double&);                  // prototipo de la función
void polar(double, double, double&, double&);  // prototipo de la función
void mostrarlo(double, double);                 // prototipo de la función

int main()
{
  double x, y, distancia, angulo;

  obtrec(x, y);
  polar(x, y, distancia, angulo);
  mostrarlo(distancia, angulo);

  return 0;
}

void obtrec(double& x, double& y)
{
  cout << "Programa de conversión de coordenadas"
       << " rectangulares a polares\n" << end;
  cout << "Introduzca la coordenada x: ";
  cin  >> x;
  cout << "Introduzca la coordenada y: ";
  cin  >> y;

  return;
}
```

(Continúa)

(Continuación)

```
void polar(double x, double y, double& r, double& theta)
{
  const double A_GRADOS = 180.0/3.141593;

  r = sqrt(x * x + y * y);
  theta = atan(y/x) * A_GRADOS;

  return;
}

void mostrarlo(double radio, double angulo)
{
  cout << "\nLas coordenadas polares son: " << endl;
  cout << " Distancia desde el origen: " << radio << endl;
  cout << " Ángulo (en grados) respecto al eje x: " << angulo << endl;

  return;
}
```

La siguiente salida se produjo con una ejecución utilizando el programa 6.14:

```
Programa de conversión de coordenadas rectangulares
a polares

Introduzca la coordenada x: 3
Introduzca la coordenada y: 4

Las coordenadas polares son:
  Distancia desde el origen: 5
  Ángulo (en grados) respecto al eje x: 53.1301
```

Antes de dejar el programa 6.14, hay que señalar que una alternativa al escribir programas controladores para cada subrutina conforme se desarrollan las subrutinas es escribir primero un programa `main()` y agregar después las subrutinas conforme se desarrollan. Esto se logra utilizando cabos para cada función (véase la sección 6.1) y luego reemplazando cada cabo, uno a la vez, con la función completada.

Aplicación 2: Simulación

Hay muchos problemas de simulación científicos y de ingeniería en los que hay que considerar la probabilidad o deben utilizarse técnicas de muestreo estadístico. Por ejemplo, al simular el flujo de tráfico de automóviles o los patrones de uso de teléfono, se requieren modelos estadísticos. Además, aplicaciones como juegos de computadora simples y escenarios más vinculados con la ingeniería sólo pueden describirse en forma estadística. Todos estos modelos estadísticos requieren la **generación de números** aleatorios; es decir, una serie de números cuyo orden no pueda predecirse.

En la práctica, no hay números que en verdad sean aleatorios. Los dados nunca son perfectos; las cartas nunca se barajan en forma completamente aleatoria; los movimientos de las moléculas que se suponen aleatorios son influidos por el ambiente; y las computado-

ras digitales sólo pueden manejar números dentro de un rango finito y con precisión limitada. Lo mejor que puede hacerse es generar números ***seudoaleatorios,*** los cuales son suficientemente aleatorios para la tarea a realizar.

Algunos lenguajes de computadora contienen una función de biblioteca que produce números aleatorios; otros no. Las funciones proporcionadas por C++ se llama `rand()`, la cual genera números aleatorios, y `srand()`, la cual establece valores "semilla" aleatorios iniciales. Se presentarán estas dos funciones y luego se usarán en una aplicación que simula el lanzamiento de una moneda para determinar el número de caras y cruces resultante.

Generación de números seudoaleatorios

Los compiladores de C++ proporcionan dos funciones para crear números aleatorios: `rand()` y `srand()`. La función `rand()` produce una serie de números aleatorios en el rango $0 \leq$ `rand()` $\leq$ `RAND_MAX`, donde la constante `RAND_MAX` se define en el archivo de encabezado `cmath`. La función `srand()` proporciona un valor "semilla" inicial para `rand()`. Si no se usa `srand()` o alguna otra técnica de "sembrado" equivalente, `rand()` producirá siempre la misma serie de números aleatorios.

El procedimiento general para crear una serie de *n* números aleatorios usando funciones de biblioteca de C++ se ilustra con el siguiente código:

```
srand(time(NULL)); // esto genera el primer valor "semilla"

for (int i = 1; i <= N; i++) // esto genera N números aleatorios
{
  rvalor = rand();
  cout << rvalor << endl;
}
```

Aquí, el argumento para la función `srand()` es una llamada a la función `time()` con un argumento `NULL`. Con este argumento la función `time()` lee el tiempo del reloj interno de la computadora, en segundos. Entonces la función `srand()` usa este tiempo, convertido a un `int` sin signo para inicializar la función generadora de números aleatorios `rand()`.[14] El programa 6.15 usa este código para generar una serie de 10 números aleatorios.

[14]De manera alternativa, muchos compiladores de C++ tienen una rutina `randomize()` que se define usando la función `srand()`. Si esta rutina está disponible, la llamada `randomize()` puede usarse en lugar de la llamada `srand(time(NULL))`. En cualquier caso, la rutina "semilla" de inicialización es llamada una sola vez, después de lo cual se usa la función `rand()` para generar una serie de números.

Programa 6.15

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <ctime>
using namespace std;

// este programa genera diez números seudoaleatorios
// usando la funcion rand() de C++

int main()
{
  const int NUMEROS = 10;

  double valor_azar;
  int i;

  srand(time(NULL));
  for (i = 1; i <= NUMEROS; i++)
  {
    valor_azar = rand();
    cout << valor_azar << endl;
  }

  return 0;
}
```

La siguiente es la salida producida por una ejecución del programa 6.15:

```
20203
21400
15265
26935
 8369
10907
31299
15400
 5074
20663
```

Debido a la llamada a la función `srand()` en el programa 6.15, la serie de diez números aleatorios diferirá cada vez que se ejecute el programa. Sin el efecto de "sembrado" aleatorizador de esta función, siempre se produciría la misma serie de números aleatorios.

Nótese también la inclusión de los archivos de encabezado `cmath` y `ctime`. El archivo cmath contiene los prototipos de función para las funciones `srand()` y `rand()`, mientras el archivo de encabezado `ctime` contiene el prototipo de función para la función `time()`.

Escalamiento

En la práctica por lo general debe hacerse una modificación al número aleatorio producido por la función `rand()`. En la mayor parte de las aplicaciones, se requiere que los números aleatorios sean valores de punto flotante dentro del rango 0.0 a 1.0 o que sean números enteros dentro de un rango especificado, como 1 a 100. El método para ajustar los números aleatorios producidos por un generador de números aleatorios para que residan dentro de dichos rangos se llama **escalamiento.**

El escalamiento de números aleatorios para que residan dentro del rango 0.0 a 1.0 se logra con facilidad dividiendo el valor devuelto de `rand()` entre `RAND_MAX.` Por tanto, la expresión `double (rand())/RAND_MAX` produce un número aleatorio en precisión doble entre 0.0 y 1.0.

El escalamiento de un número aleatorio como un valor entero entre 0 y *N*-1 se logra usando cualquiera de las expresiones `rand() % N` o `int(double(rand())/RAND_MAX * N)`. Por ejemplo, la expresión `int(double(rand())/RAND_MAX * 100)` produce un número entero aleatorio entre 0 y 99.

Para producir un número entero aleatorio entre 1 y *N* puede usarse la expresión `1 + rand() % N`. Por ejemplo, al simular el lanzamiento de un dado, la expresión `1 + rand() % 6` produce un número entero aleatorio entre 1 y 6. Puede usarse la expresión de escalamiento más general `a + rand() % (b + 1 – a)` para producir un número entero aleatorio entre los números *a* y *b*.

Un uso común de los números aleatorios es simular eventos usando un programa, en lugar de dedicar tiempo y costo a construir un experimento real. Por ejemplo, la teoría estadística indica que la probabilidad de obtener cara en un solo lanzamiento de una moneda es $\frac{1}{2}$. Del mismo modo, hay una probabilidad de 50% de obtener cruz en un solo lanzamiento de una moneda.

Usando estas probabilidades se esperaría que una sola moneda que es lanzada 1000 veces produzca 500 caras y 500 cruces. En la práctica, sin embargo, esto no se realiza nunca con exactitud en un solo experimento consistente de 1000 lanzamientos. En lugar de lanzar en realidad una moneda 1000 veces, podemos usar un generador de números aleatorios para simular esos lanzamientos.

Paso 1 *Analizar el problema*

Para este problema se requieren dos salidas: el porcentaje de caras y el porcentaje de cruces que resultan cuando se simula que una moneda es lanzada 1000 veces. No se requerirá ningún elemento de entrada para la función del generador de números aleatorios.

Paso 2 Desarrollar una solución

El porcentaje de caras y cruces se determina como

$$\text{percentaje de caras} = \frac{\text{número de caras}}{1000} \times 100\%$$

$$\text{percentaje of cruces} = \frac{\text{número de cruces}}{1000} \times 100\%$$

Para determinar el número de caras y cruces, se tienen que simular 1000 números aleatorios de tal manera que se pueda definir un resultado de "caras" o "cruces" de cada número generado. Hay diversas formas de hacer esto.

Una forma es usar la función `rand()` para generar números enteros entre 0 y `RAND_MAX`. Sabiendo que cualquier lanzamiento individual tiene una probabilidad de 50% de ser cara o cruz, podría designarse una "cara" como un número aleatorio par y una "cruz" como un número aleatorio non. Otro método sería escalar el valor devuelto de `rand()` para que resida entre 0.0 y 1.0 como se describió antes. Entonces se podría definir una "cara" como cualquier número mayor que 0.5 y cualquier otro resultado como una "cruz". Éste es el algoritmo que se adoptará.

Habiendo definido cómo se creará un solo lanzamiento que tiene una probabilidad de 50% de producir caras o cruces, la generación de 1000 lanzamientos es bastante simple: se usa un ciclo de cuenta fija que genera 1000 números aleatorios. Para cada generación se identifica el resultado como cara o cruz y se acumulan los resultados en un contador de caras y cruces. Por tanto, el algoritmo de simulación completo está dado por el seudocódigo

```
Inicializar un contador de caras en cero
Inicializar un contador de cruces en cero
For 1000 veces
  generar un número aleatorio entre 0 y 1
  If el número aleatorio es mayor que 0.5
    considérelo como cara y
    sume uno a la cuenta de caras
  Else
    considérelo cruz y
    sume uno a la cuenta de cruces
  Endif
Endfor
Calcular el porcentaje de caras como el número de caras dividido entre 1000 × 100%
Calcular el porcentaje de cruces como el número de cruces dividido entre 1000 × 100%
Imprimir el porcentaje de caras y cruces obtenido
```

Paso 3 Codificar la solución

El programa 6.16 codifica este algoritmo en C++.

Programa 6.16

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <ctime>
using namespace std;

// un programa para simular el lanzamiento de una moneda NUMLANZ veces
int main()
{
  const int NUMLANZ = 1000;

  int caras = 0; // inicializa el contador de caras
  int cruces = 0; // inicializa el contador de cruces
  int i;
  double volado, porcaras, porcruces;

    // simular los lanzamientos de NUMLANZ de una moneda
  srand(time(NULL));
  for (i = 1; i <= NUMLANZ; i++)
  {
    volado = double(rand())/RAND_MAX; // escala el número entre 0 y 1
    if (volado > 0.5)
      cabezas = cabezas + 1;
    else
      cruces = cruces + 1;
  }
  porcaras = (caras / double (NUMLANZ)) * 100.0; // calcula el porcentaje de caras
  porcruces = (cruces / double (NUMLANZ)) * 100.0; // calcula el porcentaje de cruces
  cout << "\nCaras salio " << porcaras << " por ciento de las veces";
  cout << "\nCruces salio " << porcruces << " por ciento de las veces";

  return 0;
}
```

A continuación hay dos muestras de ejecuciones usando el programa 6.16.

```
Caras salio 50.9 por ciento de las veces
Cruces salio 49.1 por ciento de las veces
```

y

```
Caras salio 49.7 por ciento de las veces
Cruces salio 50.3 por ciento de las veces
```

Por supuesto es más fácil escribir y ejecutar el programa 6.16 que lanzar una moneda 1000 veces en forma manual. Debe señalarse que la validez de los resultados producidos por el programa depende de cuán aleatorios son en realidad los números producidos por `rand()`.

Paso 4 *Probar y depurar el programa*

En realidad hay dos pruebas que el programa 6.16 debe pasar. La prueba más importante se refiere a la aleatoriedad de cada número generado. Ésta, por supuesto, es en realidad una prueba de la función de números aleatorios. Para nuestros propósitos, se ha usado una función escrita con anterioridad suministrada por el compilador. Así que en este punto aceptamos la "aleatoriedad" del generador. (Véase el ejercicio 13 para un método de verificación de la aleatoriedad de la función.)

Una vez que se ha resuelto la cuestión del generador de números aleatorios, la segunda prueba requiere que se generen en forma correcta 1000 números y acumular una cuenta de caras y cruces. Que esto se logra en forma correcta se verifica de manera adecuada con una simple verificación de escritorio del ciclo `for` dentro del programa 6.16. Además, sabemos que el resultado de la simulación debe estar cerca del 50% de caras y 50% de cruces. Los resultados de la simulación verifican que esto fue así.

Ejercicios 6.4

1. El volumen, v, y el área superficial, s, de un cilindro están dados por las fórmulas $v = \pi r^2 l$ y $s = 2\pi rl$, donde r es el radio del cilindro y l es su largo. Usando estas fórmulas, escriba y pruebe una función nombrada `cilindro()` que acepte el radio y el largo de un cilindro y devuelva su volumen y su área superficial.

2. Escriba y pruebe una función en C++ que calcule el radio, r, y el área, a, de un círculo cuando se da su circunferencia, c. Las fórmulas relevantes son $r = c/(2\pi)$ y $a = \pi r^2$. Pruebe su función usando un programa que tenga una función controladora dedicada.

3. Un líquido puede fluir por un tubo con un flujo en un patrón suave conocido como flujo *laminar* o en un patrón violento conocido como flujo *turbulento*. Las velocidades que produce cada tipo de flujo dentro del tubo pueden determinarse usando las fórmulas:

   $v_{lam} = 2100\mu / \rho\, d$ and $v_{tur} = 4000\mu / \rho\, d$,

   donde

   v_{lan} es la velocidad del líquido, en pies/s, que produce un flujo laminar determinado

   v_{tur} es la velocidad del líquido, en pies/s, que produce un flujo turbulento determinado

   μ es la viscosidad del líquido, en libras-pie por s/pie^2

   ρ es la densidad del líquido, en slug/pies^3

   d es el diámetro interior del tubo, en pies

   Usando estas fórmulas, escriba y pruebe una función en C++ llamada `flujo()` que devuelva tanto la velocidad de flujo laminar, v_{lan}, y la velocidad de flujo turbulento, v_{tur}, usando parámetros de referencia. La función deberá calcular estas velocidades para el agua, la cual tiene una viscosidad, μ, de 1.9×10^5 libras-pie por s/pie^2 y una densidad, ρ, de 1.94 slug/pies^3. El diámetro del tubo deberá ser transmitido por valor en la función `flujo()`.

4. La viscosidad y densidades de tres líquidos comunes se enumeran a continuación:

	Viscosidad (libras-pies s/pies^2)	Densidad (slug/pies^3)
Alcohol etílico	2.29×10^5	1.527
Alcohol metílico	1.17×10^5	1.531
Alcohol propílico	4.01×10^5	1.556

Usando estos datos, escriba y pruebe una función en C++ llamada `viscDen()` que devuelva la viscosidad y la densidad del líquido seleccionado usando parámetros de referencia. El tipo de líquido deberá introducirse a la función como un carácter que es transmitido por valor.

5. Escriba un programa en C++ que acepte las coordenadas rectangulares de dos puntos (x_1, y_1) y (x_2, y_2), calcule la distancia de cada punto a partir del origen y la distancia entre los dos puntos. La distancia entre dos puntos, d, está dada por la fórmula

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

6. Modifique el programa 6.16 de modo que solicite el número de lanzamientos del usuario. (*Sugerencia:* Asegúrese de hacer que el programa determine en forma correcta los porcentajes de caras y cruces obtenidas.)

7. Se han desarrollado muchos algoritmos para generar números seudoaleatorios. Algunos de estos algoritmos utilizan un esquema de conteo, como contar bits comenzando en alguna ubicación arbitraria en una memoria cambiante. Otro esquema, el cual crea números seudoaleatorios realizando un cálculo, es el método del residuo de potencias.

El **método del residuo de potencias** comienza con un número entero non de n dígitos, el cual se conoce como en número "semilla". La semilla se multiplica por el valor $(10^{n/2} - 3)$. Usar los n dígitos inferiores del resultado (el "residuo") produce una nueva semilla. Continuar este procedimiento produce una serie de números aleatorios, con cada nuevo número usado como la semilla para el siguiente número. Si la semilla original tiene cuatro o más dígitos (n igual a o mayor que 4) y no es divisible entre dos ni cinco, este procedimiento produce $5 \times 10^{(n-2)}$ números aleatorios antes que una secuencia de números se repita. Por ejemplo, si se comienza con una semilla de seis dígitos ($n = 6$), como 654321, puede generarse una serie de $5 \times 10^4 = 50\,000$ números aleatorios.

Como un algoritmo, los pasos específicos en la generación de números seudoaleatorios usando un procedimiento de residuo de potencias son los siguientes:

Paso 1: Hacer que un usuario introduzca una semilla en número entero de seis dígitos que no sea divisible entre 2 ni 5; esto significa que el número deberá ser un número non que no termine en 5.
Paso 2: Multiplicar el número semilla por 997, lo cual es $10^3 - 3$.
Paso 3: Extraer los seis dígitos inferiores del resultado producido por el paso 2. Use este número aleatorio como la siguiente semilla.
Paso 4: Repita los pasos 2 y 3 para tantos números aleatorios como sean necesarios.

Por tanto, si el número semilla introducido por el usuario es 654321 (paso 1), el primer número aleatorio generado se calcula como sigue:

Paso 2: 654321 * 997 = 652358037
Paso 3: Extraer los seis dígitos inferiores del número obtenido en el paso 2. Esto se logra usando un "truco" de programación estándar.

El truco implica:

Paso 3a: Dividir el número entre 10^6 = 1000000.
Por ejemplo, 652358037 / 1000000 = 652.358037
Paso 3b: Tomar la parte entera del resultado del paso 3a.
Por ejemplo, la parte entera de 652.358037 = 652
Paso 3c: Multiplicar el resultado anterior por 10^6
Por ejemplo, $652 \times 10^6 = 652000000$
Paso 3d: Restar este resultado del número original.
Por ejemplo, 652358037 – 652000000 = 358037

La parte entera de un número en punto flotante puede tomarse asignando el número en punto flotante a una variable en número entero, o mediante un molde de C++ (véase la sección 3.3). En nuestro procedimiento se utilizará el mecanismo de molde. Por tanto, el algoritmo para producir un número aleatorio puede lograrse usando el siguiente código:

```
i = int(997.0 * x / 1.e6);    // tomar la parte entera
x = 997.0 * x - i * 1.e6;
```

Usando esta información,

a. Cree una función llamada `numalea()` que acepte una "semilla" en punto flotante como un parámetro y devuelva un número aleatorio en punto flotante entre 0 y 1.e6.
b. Incorpore la función `numalea()` creada en el ejercicio 7a en un programa en C++ que produzca diez números aleatorios entre 0 y 1.e6.
c. Pruebe la aleatoriedad de la función `numalea()` creada en el ejercicio 7a usando el método descrito en el ejercicio 13. Inténtelo con algunos valores semilla pares y algunos valores semilla nones que terminen en 5 para determinar si esto afecta la aleatoriedad de los números.

8. Escriba una función en C++ que determine en cuál cuadrante reside una línea trazada desde el origen. La determinación del cuadrante se hace usando el ángulo que forma la línea con el eje *x* positivo como sigue:

Ángulo desde el eje *x* positivo	Cuadrante
Entre 0 y 90 grados	1
Entre 90 y 180 grados	2
Entre 180 y 270 grados	3
Entre 270 y 360 grados	4

NOTA: Si el ángulo tiene exactamente 0, 90, 180 o 270 grados la línea correspondiente no reside en ningún cuadrante sino que se encuentra en un eje. Para este caso su función deberá devolver un cero.

9. Escriba un programa para simular el lanzamiento de dos dados. Si el total de los dos dados es 7 u 11 usted gana; de lo contrario pierde. Adorne este programa tanto como quiera, con apuestas, posibilidades diferentes, combinaciones diferentes para ganar o perder, dejar de jugar cuando no le quede dinero o alcance el límite de la casa, desplegar los dados, etc. (*Sugerencia:* Calcule los puntos que se muestran en cada dado con la expresión `puntos = (int)(6.0 * numero aleatorio + 1)`, donde el número aleatorio esté entre 0 y 1.)

10. Un valor que a veces es útil es el máximo común divisor de dos números enteros *n*1 y *n*2. Hace más de dos mil años un matemático famoso, Euclides, descubrió un método eficiente para hacer esto. Sin embargo, por ahora nos conformaremos con un cabo. Escriba la función en número entero `stub gcd(n1, n2)`. Tan sólo haga que devuelva un valor que sugiera que recibió sus parámetros en forma correcta. (*Sugerencia: n*1 + *n*2 es una buena elección de valores de devolución. ¿Por qué *n*1/*n*2 no es una buena elección?)

11. El método de Euclides para encontrar el máximo común divisor (GCD, por sus siglas en inglés) de dos números enteros positivos consiste en los siguientes pasos:

a. Divida el número mayor entre el menor y conserve el residuo.
b. Divida el número menor entre el residuo, conservando de nuevo el residuo.
c. Continúe dividiendo el residuo anterior entre el residuo actual hasta que el residuo sea cero, punto en el cual el último residuo diferente de cero es el máximo común divisor.

Por ejemplo, suponga que los dos números enteros positivos son 84 y 49, tenemos:

Paso a: 84/49 produce un residuo de 35
Paso b: 49/35 produce un residuo de 14
Paso c: 35/14 produce un residuo de 7
Paso d: 14/7 produce un residuo de 0

Por tanto, el último residuo diferente de cero, el cual es 7, es el máximo común divisor de 84 y 49.

Utilizando el algoritmo de Euclides, reemplace la función cabo escrita para el ejercicio 10 con una función real que determine y devuelva el GCD de sus dos parámetros en número entero.

12. El siguiente programa usa los mismos nombres de variables tanto en la función que llama como en la función llamada. Determine si esto causa algún problema para el compilador.

```
#include <iostream.h>

int tiempo(int, int); // prototipo de la función

int main()
```

```
{
  int min, hora, seg;

  cout << "Introduzca dos números: ";
  cin >> min, hora;
  seg = tiempo(min, hora);
  cout << "El número total de segundos es " << seg <<
endl;

  return 0;
}

int tiempo(int min, int hora)
{
  int seg;

  seg = (hora * 60 + min) * 60;
  return seg;
}
```

13. Escriba un programa que pruebe la efectividad de la función de biblioteca `rand()`. Empiece inicializando en cero 10 contadores, como `cuentacero`, `cuentauno`, `cuentados`,..., `cuentanueve`. Luego genere un número grande de números enteros seudoaleatorios entre 0 y 9. Cada vez que ocurra un 0 incremente `cuentacero`, cuando ocurra un 1 incremente `cuentauno`, etc. Por último, imprima el número de 0, 1, 2, etc., que ocurrieron y el porcentaje de veces que ocurrieron.

14. El determinante de la matriz de 2 por 2

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

es $a_{11}a_{22} - a_{21}a_{12}$.

Del mismo modo, el determinante de una matriz de 3 por 3

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} =$$

$$a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{21}\begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix} + a_{31}\begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix}$$

Utilizando esta información, escriba y pruebe dos funciones, llamadas `det2()` y `det3()`. La función `det2()` deberá aceptar los cuatro coeficientes de una matriz de 2 por 2 y devolver su determinante. La función `det3()` deberá aceptar los nueve coeficientes de una matriz de 3 por 3 y devolver su determinante llamando a `det2()` para calcular los determinantes 2 por 2 requeridos.

6.5 ALCANCE DE UNA VARIABLE

Ahora que hemos comenzado a escribir programas que contienen más de una función, se pueden observar con más detenimiento las variables declaradas dentro de cada función y su relación con las variables en otras funciones.

Por su misma naturaleza, las funciones en C++ están construidas como módulos independientes. Como se ha visto, los valores se transmiten a la función usando la lista de parámetros de la función y se devuelve un valor usando una instrucción de devolución. Visto desde esta perspectiva, una función puede considerarse como una caja cerrada, con ranuras en la parte superior para recibir valores y una sola ranura en la parte inferior de la caja para devolver un valor (véase la figura 6.15).

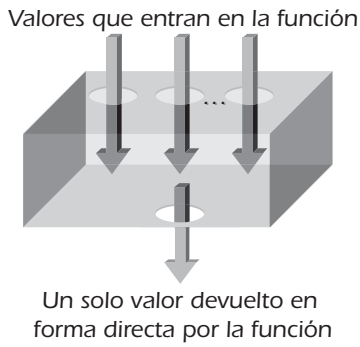


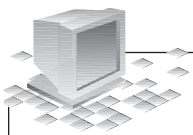
Figura 6.15 *Una función puede considerarse una caja cerrada.*

La metáfora de una caja cerrada es útil porque enfatiza el hecho que todo lo que va dentro de la función, incluyendo todas las declaraciones de variables dentro del cuerpo de la función, está oculto de la vista de todas las demás funciones. Dado que las variables creadas dentro de una función están disponibles de manera convencional sólo para la función en sí, se dice que son locales para la función o ***variables locales***. Este término se refiere al alcance de un identificador, donde ***alcance*** se define como la sección del programa donde el identificador, como una variable, es válido o "conocido". Se dice que esta sección del programa es donde la variable es visible. Una variable puede tener ***alcance*** local o global. Una variable con un ***alcance local*** es aquella a la que una instrucción de declaración hecha dentro del cuerpo de una función le ha designado ubicaciones de almacenamiento. Las variables locales sólo son significativas cuando se usan en expresiones o instrucciones dentro de la función que las declaró. Esto significa que el mismo nombre de variable puede declararse y usarse en más de una función. Para cada función que declara la variable, se crea una variable separada y distinta.

Todas las variables que se han utilizado hasta ahora han sido variables locales. Éste es un resultado directo de colocar las instrucciones de declaración dentro de las funciones y usarlas como instrucciones de definición que causan que la computadora reserve almacenamiento para la variable declarada. Como se verá, las instrucciones de declaración pueden colocarse fuera de las funciones y no necesitan actuar como definiciones para que se reserven nuevas áreas de almacenamiento para la variable declarada.

Una variable con ***alcance global,*** por lo general denominada ***variable global,*** el almacenamiento se crea mediante una instrucción de declaración localizada fuera de cualquier

función. Estas variables pueden ser utilizadas por todas las funciones que se colocan físicamente después de la declaración de la variable global. Esto se muestra en el programa 6.17, donde se utiliza a propósito el mismo nombre de variable dentro de ambas funciones contenidas en el programa.



Programa 6.17

```
#include <iostream>
using namespace std;

int primernum; // crea una variable global llamada primernum

void valfun(); // prototipo de la función (declaración)

int main()
{
  int segundonum;          // crea una variable local llamada segundonum

  primernum = 10; // almacena un valor en la variable global
  segundonum = 20; // almacena un valor en la variable local

  cout << "De main(): primernum = " << primernum << endl;
  cout << "De main(): segundonum = " << segundonum << endl;

  valfun(); // llama a la función valfun

  cout << "\nDe main() de nuevo: primernum = " << primernum << endl;
  cout << "De main() de nuevo: segundonum = " << segundonum << endl;

  return 0;
}

void valfun() // no se transmiten valores a esta función
{
  int segundonum; // crea una segunda variable local llamada segundonum

  segundonum = 30; // esto sólo afecta al valor de esta variable local

  cout << "\nDe valfun(): primernum = " << primernum << endl;
  cout << "De valfun(): segundonum = " << segundonum << endl;

  primernum = 40; // esto cambia primernum para ambas funciones

  return;
}
```

La variable `primernum` en el programa 6.17 es una variable global debido a que su almacenamiento fue creado por una instrucción de definición localizada fuera de una función. En vista que ambas funciones, `main()` y `valfun()`, siguen a la definición de `primernum`, ambas funciones pueden usar esta variable global sin que necesiten ninguna otra declaración.

El programa 6.17 también contiene dos variables locales separadas, ambas nombradas `segundonum`. El almacenamiento para la variable `segundonum` nombrada en `main()` es creado por la instrucción de definición localizada en `main()`. Un área de almacenamiento diferente para la variable `segundonum` en `valfun()` es creada por la instrucción de definición localizada en la función `valfun()`. La figura 6.16 ilustra las tres áreas de almacenamiento distintas reservadas por las tres instrucciones de definición que se encuentran en el programa 6.17.

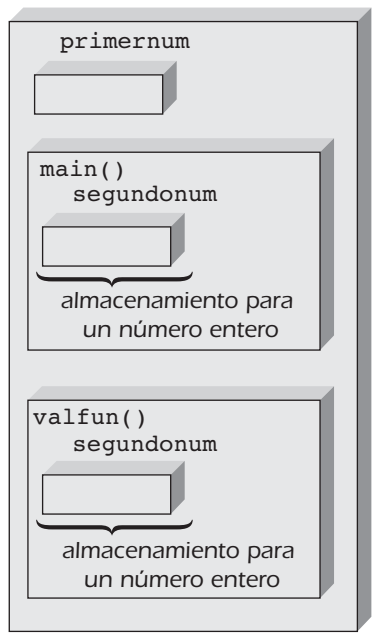


Figura 6.16 *Las tres áreas de almacenamiento creadas por el programa 6.17.*

Cada una de las variables nombradas `segundonum` es local para la función en la que se crea su almacenamiento, y cada una de estas variables sólo puede usarse desde dentro de la función apropiada. Por tanto, cuando se usa `segundonum` en `main()`, se tiene acceso al área de almacenamiento reservada por `main()` para su variable segundonum, y cuando se usa `segundonum` en `valfun()`, se tiene acceso al área de almacenamiento reservada por `valfun()` para su variable `segundonum`. La siguiente salida se produce cuando se ejecuta el programa 6.17:

```
De main(): primernum = 10
De main(): segundonum = 20

De valfun(): primernum = 10
De valfun(): segundonum = 30
```

```
De main() de nuevo: primernum = 40
De main() de nuevo: segundonum = 20
```

Vamos a analizar esta salida. En vista que `primernum` es una variable global, las funciones `main()` y `valfun()` pueden usar y cambiar su valor. Al principio, ambas funciones imprimen el valor de 10 que almacenó `main()` en `primernum`. Antes de devolver, `valfun()` cambia el valor de primernum a 40, que es el valor que corresponde cuando a continuación la variable `primernum` se despliega desde dentro de `main()`.

Debido a que cada función sólo "conoce" sus propias variables locales, `main()` sólo puede enviar el valor de su `segundonum` al objeto `cout`. Por tanto, siempre que se obtiene `segundonum` desde `main()` se despliega el valor 20, y siempre que se obtiene `segundonum` desde `valfun()` se despliega el valor 30.

C++ no confunde las dos variables `segundonum` porque sólo puede ejecutarse una función en un momento determinado. Mientras se ejecuta una función, sólo se puede tener acceso a aquellas variables y parámetros que están "al alcance" de esa función (global y local).

El alcance de una variable de ninguna manera influye o restringe el tipo de datos de la variable. Del mismo modo en que una variable local puede ser un carácter, número entero, booleano, de precisión doble o cualquiera de los otros tipos de datos (largo/corto) que se han introducido, así también pueden ser las variables globales de estos tipos de datos, como se ilustra en la figura 6.17. El alcance de una variable es determinado por la colocación de la instrucción de definición que reserva almacenamiento para ella y de manera opcional por una instrucción de declaración que la hace visible, mientras el tipo de datos de la variable se determina usando la palabra clave apropiada (`char`, `int`, `bool`, `double`, etc.) antes del nombre de la variable en una instrucción de declaración.

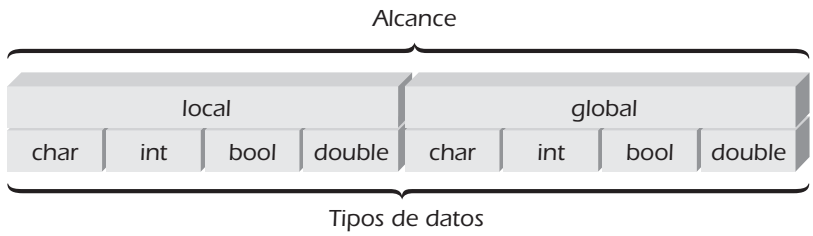


Figura 6.17 *Relación del alcance y el tipo de una variable.*

Operador de resolución de alcance

Cuando una variable local tiene el mismo nombre que una variable global, todas las referencias al nombre de la variable hechas dentro del alcance de la variable local se refieren a esta variable local. Esta situación se ilustra en el programa 6.18, donde se define el nombre de la variable `numero` como una variable global y como una local.

Programa 6.18

```
#include <iostream>
using namespace std;

double numero = 42.8;     // una variable global llamada número

int main()
{
  double numero = 26.4;     // una variable local llamada número

  cout << "El valor de número es " << número << endl;

  return 0;
}
```

Cuando se ejecuta el programa 6.18, se despliega la siguiente salida.

```
El valor de número es 26.4
```

Como lo muestra esta salida, el nombre de la variable local toma precedencia sobre la variable local. En tales casos, aún se puede tener acceso a la variable global utilizando el operador de resolución de alcance de C++. Este operador, el cual tiene el símbolo **::**, debe colocarse inmediatamente antes del nombre de la variable, como en **::** número. Cuando se usa de esta manera, **::** le indica al compilador que use la variable global. Como un ejemplo, se usa el operador de resolución de alcance en el programa 6.18a.

Program 6.18a

```
#include <iostream>
using namespace std;

double numero = 42.5;       // una variable global nombrada número

int main()
{
  double numero = 26.4;       // una variable local nombrada número

  cout << "El valor de número es " << ::número << endl;

  return 0;
}
```

La salida producida por el programa 6.18a es:

```
El valor de número es 42.5
```

Como lo indica esta salida, el operador de resolución de alcance causa que se tenga acceso a la variable global en lugar de a la local.

Mal uso de las variables globales

Las variables globales permiten al programador "brincarse" las salvaguardas normales proporcionadas por las funciones. En lugar de transmitir variables a una función, es posible hacer que todas las variables sean globales. **No haga esto.** Al hacer todas las variables globales de manera indiscriminada, al instante destruye las salvaguardas que C++ proporciona para hacer que las funciones sean independientes y aisladas entre sí, incluyendo la necesidad de designar con cuidado el tipo de argumentos que necesita una función, las variables usadas en la función y el valor devuelto.

Usar sólo variables globales puede ser desastroso, en especial en programas más grandes que tienen muchas funciones creadas por el usuario. En vista que todas las variables en una función deben ser declardas, crear funciones que utilicen variables globales requiere recordar escribir las declaraciones globales apropiadas al principio de cada programa que use la función; ya no se encuentran junto con la función. Sin embargo, más devastador que esto es tratar de rastrear un error en un programa grande que utiliza variables globales. Dado que cualquier función que siga a la declaración global puede tener acceso a una variable global y cambiarla, localizar el origen de un valor erróneo es una tarea frustrante y que consume tiempo.

Sin embargo, las definiciones globales, en ocasiones son útiles para crear variables y constantes que deben ser compartidas entre muchas funciones. En lugar de transmitir la misma variable a cada función, es más fácil definir la variable una vez como global. Hacer esto alerta a cualquiera que lea el programa respecto a que muchas funciones usan esta variable. La mayor parte de los programas grandes casi siempre usan unas cuantas variables o constantes globales. Sin embargo, los programas más pequeños que contienen sólo algunas funciones casi nunca deben contener variables globales.

El mal uso de las variables globales no se aplica a los prototipos de funciones, los cuales por lo general son globales. Debe señalarse que todos los prototipos de función que hasta ahora se han usado han sido de alcance global, declaran el prototipo para todas las funciones subsiguientes. Colocar un prototipo de función dentro de una función hace del prototipo una declaración local disponible sólo para la función que se declara adentro.

Ejercicios 6.5

1. a. Para la siguiente sección de código, determine el tipo de datos y el alcance de todas las funciones declaradas. Para hacer esto, utilice una hoja de papel separada y anote los tres encabezados de columna que siguen (se han llenado las entradas para la primera variable):

Nombre del a variable	Tipo de datos	Alcance
`voltios`	`int`	`global para main, roi y paso`

```
#include <iostream>
using namespace std;

int voltios;
long int resistencia;
double corriente;

int main()
{
  int potencia;
  double factor, tiempo;
     .
     .
  return 0;
}

double roi(int mat1, int mat2)
{
  int cuenta;
  double peso;
       .
       .
  return peso;
}

int paso(double primero, double ultimo)
{
  int horas;
  double partefrac;
       .
       .
  return 10*horas;
}
```

b. Dibuje cuadros alrededor de la sección apropiada del código anterior para encerrar el alcance de cada variable.

c. Determine el tipo de datos de los parámetros que esperan las funciones `roi` y `paso`, y el tipo de datos del valor devuelto por estas funciones.

2. a. Para la siguiente sección de código, determine el tipo de datos y el alcance de todas las variables declaradas. Para hacer esto, use una hoja de papel separada y anote los tres encabezados de columna que siguen (se han llenado las entradas de la primera variable):

Nombre de la variable	Tipo de datos	Alcance
`clave`	`char`	`global para main, func1 y func2`

```
#include <iostream>
using namespace std;

char clave;
long int numero;

int main()
{
  int a,b,c;
  double x,y;
     .
     .
  return 0;
}

double segundonum;

int func1(int num1, int num2)
{
  int o,p;
  float q;
     .
     .
  return p;
}

double func2(double primero, double ultimo)
{
  int a,b,c,o,p;
  double r;
  double s,t,x;
     .
     .
  return s * t;
}
```

b. Dibuje un cuadro alrededor de la sección apropiada del código anterior para encerrar el alcance de las variables `clave`, `segundonum`, `y` y `r`.

c. Determine el tipo de datos de los argumentos que esperan las funciones `func1` y `func2`, y el tipo de datos del valor devuelto por estas funciones.

3. Además de hablar sobre el alcance de una variable, también se puede aplicar el término a los parámetros de una función. ¿Cuál piensa que es el alcance de todos los parámetros de función?

4. Considere la siguiente estructura de programa:

```
#include <iostream>
using namespace std;

int a, b;
double Uno(float);
void Dos(void);
int main()
{
  int c, d;
  double e, f;
    .
    .
  return 0;
}

double Uno(double p2)
{
  char m, n;
    .
    .
}

void Dos(void)
{
  int p, d;
  double q, r;
    .
    .
}
```

Defina el alcance del parámetro p2 y las variables `a`, `b`, `c`, `d`, `m`, `n`, `p`, `d`, `q` y `r`.

5. Determine los valores desplegados por cada instrucción `cout` en el siguiente programa:

```
#include <iostream>
using namespace std;

int primernum = 10; // declara e inicializa una variable
global
void desplegar(); // prototipo de la función
int main()
{
  int primernum = 20; // declara e inicializa una variable
  local
  cout << "\nEl valor de primernum es " << primernum << endl;
  desplegar();

  return 0;
}
```

```
void desplegar(void)
{
  cout << "Ahora el valor de primernum es " << primernum
  << endl;
  return;
}
```

6.6 CLASES DE ALMACENAMIENTO DE VARIABLES

El alcance de una variable define la ubicación dentro de un programa donde ésta puede usarse. Dado un programa, se podría tomar un lápiz y trazar un cuadro alrededor de la sección del programa donde es válida cada variable. El espacio dentro del cuadro representaría el alcance de una variable. Desde este punto de vista, el alcance de una variable puede considerarse como el espacio dentro del programa donde es válida la variable.

Además de la dimensión espacial representada por su alcance, las variables también tienen una dimensión temporal. La dimensión temporal se refiere al tiempo que las ubicaciones de almacenamiento son reservadas para una variable. La dimensión temporal se conoce como la "vida" de la variable. Por ejemplo, todas las ubicaciones de almacenamiento de la variable se devuelven a la computadora cuando un programa termina su ejecución. Sin embargo, mientras un programa aún está en ejecución, las áreas de almacenamiento de la variable provisional se reservan y en lo subsiguiente se devuelven a la computadora. Puede determinarse dónde y cuánto se conservan las áreas de almacenamiento de una variable antes que sean liberadas por la **clase de almacenamiento** de la variable.

Además de tener un tipo de datos y un alcance, cada variable tiene también una clase de almacenamiento. Las cuatro clases de almacenamiento disponibles se llaman `auto`, `static`, `extern` y `register`. Si se usa uno de estos nombres de clase, debe colocarse antes del tipo de datos de la variable en una instrucción de declaración. Son ejemplos de instrucciones de declaración que incluyen una designación de clase de almacenamiento:

```
auto int num;           // clase de almacenamiento auto y tipo de datos int
static int millas;      // clase de almacenamiento static y tipo de datos int
register int dist;      // clase de almacenamiento register y tipo de datos int
extern int voltios;     // clase de almacenamiento extern y tipo de datos int
auto float cupon;       // clase de almacenamiento auto y tipo de datos float
static double años;     // clase de almacenamiento static y tipo de datos double
extern float resultado; // clase de almacenamiento extern y tipo de datos float
auto char tecla_en;     // clase de almacenamiento auto y variable char
```

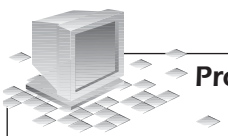
Para entender lo que significa la clase de almacenamiento de una variable, se considerarán primero las variables locales (aquellas variables creadas dentro de una función) y luego las variables globales (aquellas variables creadas fuera de una función).

Clases de almacenamiento de variables locales

Las variables locales sólo pueden ser miembros de las `clases de almacenamiento auto, static o register.` Si no se incluye una descripción de clase en la instrucción de declaración, la variable es asignada de manera automática a la categoría `auto`. Por tanto, `auto`

es la clase por omisión usada por C++. Todas las variables locales que se han usado, en vista que se omitió la designación de la clase de almacenamiento, han sido variables `auto`.

El término `auto` es una abreviatura de ***automático***. El almacenamiento para variables automáticas locales es reservado o creado en forma automática cada vez que se llama una función que declara variables automáticas. En tanto la función no haya regresado el control a la función que la llama, todas las variables automáticas locales para la función están "vivas"; es decir, el almacenamiento de las variables está disponible. Cuando la función devuelve el control a la función que la llama, sus variables automáticas locales "mueren"; es decir, el almacenamiento para las variables es devuelto a la computadora. Este proceso se repite cada vez que es llamada una función. Por ejemplo, considere el programa 6.19, donde se llama tres veces a la función `probarauto()` desde `main()`.



Programa 6.19

```
#include <iostream>
using namespace std;

void probarauto();    // prototipo de la función

int main()
{
  int cuenta;              // cuenta es una variable automática local

  for(cuenta = 1; cuenta <= 3; cuenta++)
    probarauto();

  return 0;
}

void probarauto()
{
  int num = 0;     // num es una variable automática local
                   // inicializada en cero
  cout << "El valor de la variable automática num es "
       << num << endl;
  num++;

  return;
}
```

La salida producida por el programa 6.19 es:

```
El valor de la variable automática num es 0
El valor de la variable automática num es 0
El valor de la variable automática num es 0
```

Cada vez que es llamada `probarauto()`, se crea e inicializa en cero la variable automática `num`. Cuando la función devuelve el control a `main()`, la variable `num` se destruye junto con cualquier valor almacenado en `num`. Por tanto, se pierde el efecto de incrementar num en `probarauto()`, antes de la instrucción de devolución de la función, cuando se regresa el control a `main()`.

Para la mayor parte de las aplicaciones, el uso de variables automáticas funciona bien. Sin embargo, hay casos en los que nos gustaría que una función recordara valores entre las llamadas a la función. Éste es el propósito de la clase de almacenamiento `static`. Una vez creadas, las variables `static` locales permanecen en existencia durante la vida del programa. Esto significa que el último valor almacenado en la variable cuando termina de ejecutarse la función está disponible para la función la próxima vez que es llamada.

```
static int rate;
static double resistance;
static char inKey;
```

Una variable local `static` no se crea y se destruye cada vez que se llama a la función que declara dicha variable. Una vez creadas, las variables locales `static` siguen existiendo durante la vida del programa. Esto significa que el valor menor almacenado en la variable cuando termina la función, queda disponible cuando la función vuelve a ser llamada.

Debido a que las variables `static` locales conservan sus valores, no se inicializan dentro de una instrucción de declaración en la misma forma que las variables automáticas. Para entender por qué, considérese la declaración automática `int num = 0;` la cual causa que se cree la variable automática `num` y se coloque en cero cada vez que se encuentra la declaración. Esto se llama ***inicialización en tiempo de ejecución*** porque la inicialización ocurre cada vez que se encuentra la instrucción de declaración. Este tipo de inicialización sería desastroso para una variable `static`, porque reestablecer el valor de la variable en cero cada vez que se llama la función destruiría el valor mismo que se está tratando de guardar.

La inicialización de las variables `static` (tanto locales como globales) sólo se hace una vez, cuando el programa se compila por primera vez. En tiempo de compilación se crea la variable y se coloca cualquier valor de inicialización en ella.[15] A partir de entonces, el valor en la variable es conservado sin mayor inicialización. Para ver cómo funciona esto, considere el programa 6.20.

[15] Algunos compiladores inicializan las variables static locales la primera vez que se ejecuta la instrucción de definición en vez de hacerlo cuando se compila el programa.

Programa 6.20

```
#include <iostream>
using namespace std;

void probarstat();    // prototipo de la función

int main()
{
  int cuenta;              // cuenta es una variable auto local

  for(cuenta = 1; cuenta <= 3; cuenta++)
    probarstat();

  return 0;
}

void probarstat()
{
  static int num = 0;     // num es una variable static local
  cout << "El valor de la variable static num ahora es "
       << num << endl;
  num++;

  return;
}
```

La salida producida por el programa 6.20 es

```
El valor de la variable static num ahora es 0
El valor de la variable static num ahora es 1
El valor de la variable static num ahora es 2
```

Como lo ilustra esta salida, la variable `static num` se pone en cero sólo una vez. Luego la función `probarstat()` incrementa esta variable justo antes de regresar el control a `main()`. El valor que tenía `num` cuando deja la función `probarstat()` es conservado y desplegado cuando se vuelve a llamar a la función.

A diferencia de las variables automáticas que pueden ser inicializadas por constantes o por expresiones que usan tanto constantes como variables inicializadas con anterioridad, las variables `static` sólo pueden inicializarse usando constantes o expresiones constantes, como `3.2 + 8.0`. Además, a diferencia de las variables automáticas, todas las variables `static` se colocan en cero cuando no se da una inicialización explícita. Por tanto, en el programa 6.19 no se requiere la inicialización específica de `num` en cero.

La clase de almacenamiento restante disponible para variables locales, la clase `register`, no se usa en forma tan extensa como las `variables auto` o `static`. Son ejemplos de declaraciones de variable `register`

```
register int tiempo;
register double diferencia;
register float cupon;
```

Las variables `register` tienen la misma duración que las variables `auto`; es decir, una variable `register` local se crea cuando se introduce la función que la declara y se destruye cuando la función completa su ejecución. La única diferencia entre las variables `register` y `auto` es dónde se localiza el almacenamiento para la variable.

El almacenamiento para todas las variables (locales y globales), con excepción de las variables `register`, se reserva en el área de memoria de la computadora. La mayor parte de las computadoras tienen unas cuantas áreas de almacenamiento de alta velocidad adicionales localizadas en forma directa en la unidad de procesamiento de la computadora que también pueden usarse para almacenamiento de variables. Estas áreas de almacenamiento de alta velocidad especiales se llaman **registros**. Dado que los registros se localizan físicamente en la unidad de procesamiento de la computadora, también puede tenerse acceso a ellos con más rapidez que a las áreas de almacenamiento de memoria normales localizadas en la unidad de memoria. Además, las instrucciones de computadora que hacen referencia a registros por lo general requieren menos espacio que las instrucciones que hacen referencia a ubicaciones de memoria debido a que hay menos registros a los que se pueda tener acceso que ubicaciones de memoria. Cuando el compilador sustituye la ubicación de un registro por una variable durante la compilación del programa, se necesita menos espacio en la instrucción del que se requiere para dirigirse a una memoria que tiene millones de ubicaciones.

Además de disminuir el tamaño de un programa en C++ compilado, usar variables `register` puede aumentar la velocidad de ejecución de un programa en C++ si su computadora admite este tipo de datos. Los programas de aplicaciones que están pensados para ser ejecutados en varias computadoras no deberán usar registros. Los intentos de hacerlo por lo general serán frustrados por el compilador cambiando de manera automática las variables declaradas con la clase de almacenamiento `register` a una clase de almacenamiento `auto`.

La única restricción al usar la clase de almacenamiento `register` es que no se puede utilizar la dirección de una variable `register` con el operador de direccionamiento `&`. Esto se comprende con facilidad cuando se recuerda que los registros no tienen direcciones de memoria estándares.

Clases de almacenamiento de variables globales

Las variables globales son creadas por instrucciones de definición externas a una función. Por su naturaleza, estas variables definidas en forma externa no llegan y se van con la llamada de cualquier función. Una vez que se crea una variable global, existe hasta que el programa en el que se declaró termina de ejecutarse. Por tanto, las variables globales no pueden declararse como variables `auto` o `register` que se crean y destruyen conforme se ejecuta el programa. Además, las variables globales pueden declararse como miembros de las clases de almacenamiento `static` o `extern` (pero no de ambas). Son ejemplos de instrucciones de declaración que incluyen estas dos descripciones de clases

```
extern int suma;
extern double voltios;
static double corriente;
```

Las clases `static` y `extern` sólo afectan el alcance, no la duración, de variables globales. Como con las variables `static` locales, todas las variables globales se inicializan en cero en tiempo de compilación.

El propósito de la clase de almacenamiento `extern` es extender el alcance de una variable global más allá de sus límites normales. Para entender esto, primero hay que observar que todos los programas que se han escrito hasta ahora siempre se habían contenido en un solo archivo. Por tanto, cuando guardaba o recuperaba programas sólo necesitaba darle a la computadora un nombre para su programa. Esto no se requiere en C++.

Los programas más grandes por lo general consisten en muchas funciones que son almacenadas en múltiples archivos. Un ejemplo de esto se muestra en la figura 6.18, donde las tres funciones `main()`, `func1()` y `func2()` se almacenan en un archivo y las dos funciones `func3()` y `func4()` se almacenan en un segundo archivo.

archivo1

```
int voltios;
double corriente;
static double potencia;
     .
     .
     .
int main( )
{
     func1( );
     func2( );
     func3( );
     func4( );
}
int func1( )
{
     .
     .
     .
}
int func2( )
{
     .
     .
     .
}
```

archivo2

```
double factor;
int func3( )
{
     .
     .
     .
}
int func4( )
     .
     .
     .
}
```

Figura 6.18 *Un programa puede estar almacenado en más de un archivo.*

Para los archivos ilustrados en la figura 6.18, las variables globales `voltios`, `corriente` y `potencia` declaradas en archivo1 sólo pueden ser usadas por las funciones `main()`, `func1()` y `func2()` en este archivo. La variable global única, factor, declarada en archivo2 sólo puede ser usada por las funciones `func3()` y `func4()` en el archivo2.

Aunque la variable `voltios` se ha creado en el archivo1, se puede utilizar en el archivo2. Colocar la instrucción de declaración `extern int voltios`; en el archivo2, como se muestra en la figura 6.19, nos permite hacerlo. Poner esta instrucción en la parte superior de archivo2 extiende el alcance de la variable `voltios` al archivo2 de modo que pueda usarse tanto en `func3()` como en `func4()`. Por tanto, la designación `extern` tan sólo declara una variable global que es definida en otro archivo. Así, colocar la instrucción `extern double corriente`; en `func4()` extiende el alcance de esta variable global, creada en archivo1, a la `func4()`, y el alcance de la variable global `factor`, creada en archivo2, se extiende a `func1()` y `func2()` por la instrucción de declaración `extern double factor`; colocada antes de `func1()`. Hay que observar que `factor` no está disponible para `main()`.

archivo1

```
int voltios;
double curriente;
static double potencia;
     .
     .
     .
int main( )
{
   func1( );
   func2( );
   func3( );
   func4( );
}
extern double factor;
int func1( )
{
     .
     .
     .
}
int func2( )
{
     .
     .
     .
}
```

archivo2

```
double factor;
extern int voltios;
int func3( )
{
     .
     .
     .
}
int func4( )
{
   extern double corriente;
     .
     .
     .
}
```

Figura 6.19 *Extensión del alcance de variables globales.*

Una instrucción de declaración que contiene de manera específica la palabra `extern` es diferente de todas las otras instrucciones de declaración porque no origina una variable nueva reservando nuevo almacenamiento para la variable. Una instrucción de declaración `extern` tan sólo informa a la computadora que ya existe una variable global y que ahora puede usarse. El almacenamiento real para la variable debe crearse en alguna otra parte del programa usando una, y sólo una, instrucción de declaración global en la que no se ha usado

Punto de información

Clases de almacenamiento

Las variables del tipo `auto` y `register` siempre son variables locales. Sólo variables globales no estáticas pueden declararse usando la palabra clave `extern`. Hacerlo así extiende el alcance de la variable a otro archivo o función.

Hacer `static` una variable global reduce una variable privada del archivo en el que se declara. Por tanto, las variables `static` *no pueden* utilizar la palabra clave `extern`. Excepto para variables `static`, todas las variables son inicializadas cada vez que ingresan a un alcance. Las variables `static` sólo se inicializan una vez cuando se definen.

la palabra `extern.` La inicialización de la variable global puede hacerse, por supuesto, con la declaración original de la variable global. La inicialización dentro de una instrucción de declaración `extern` no se permite y causará un error de compilación.

La existencia de la clase de almacenamiento `extern` es la razón por la cual se ha tenido tanto cuidado en distinguir entre la creación y declaración de una variable. Las instrucciones de declaración que contienen la palabra `extern` no crean nuevas áreas de almacenamiento; sólo extienden el alcance de variables globales existentes.

La última clase global, las variables globales `static,` se utilizan para prevenir la extensión de una variable global a un segundo archivo. Las variables globales `static` se declaran en la misma forma que las variables locales `static`, excepto que la instrucción de declaración se coloca afuera de cualquier función.

El alcance de una variable global `static` no puede extenderse más allá del archivo en el que es declarada. Esto proporciona un grado de privacidad para las variables globales `static.` Dado que sólo son "conocidas" y sólo pueden usarse en el archivo en el que son declaradas, otros archivos no pueden tener acceso a ellas ni cambiar sus valores. Por tanto, las variables globales `static` no pueden extenderse después a un segundo archivo usando una instrucción de declaración `extern.` Intentar hacerlo producirá un error de compilación.

Ejercicios 6.6

1. **a.** Enumere las clases de almacenamiento disponibles para variables locales.
   b. Enumere las clases de almacenamiento disponibles para variables globales.
2. Describa la diferencia entre una variable local `auto` y una variable local `static.`
3. ¿Cuál es la diferencia entre las siguientes funciones?

```
void init1()
{
  static int años = 1;
  cout << "El valor de anios es " << años << endl;
  años = años + 2;
}
```

```
void init2()
{
  static int años;
  anios = 1;
  cout << "El valor de años es " << años << endl;
  años = años + 2;
}
```

4. **a.** Describa la diferencia entre una variable global `static` y una variable global `extern`.
   b. Si una variable es declarada con una clase de almacenamiento `extern`, ¿qué otra instrucción de declaración debe estar presente en alguna otra parte del programa?
5. La instrucción de declaración `static double resistencia;` puede usarse para crear ya sea una variable local o una global. ¿Qué determina el alcance de la variable `resistencia`?
6. Para las declaraciones de función y variable ilustradas en la figura 6.20, coloque una declaración `extern` para lograr de manera individual lo siguiente:
   a. Extender el alcance de la variable global `eleccion` a todo el archivo2.
   b. Extender el alcance de la variable global `marcador` sólo a la función `promedio()`.
   c. Extender el alcance de la variable global fecha a `promedio()` y `variancia()`.
   d. Extender el alcance de la variable global `fecha` sólo a `roi()`.
   e. Extender el alcance de la variable global `factor` sólo a `roi()`.
   f. Extender el alcance de la variable global `tipo_b` a todo el archivo1.
   g. Extender el alcance de la variable global `resistencia` a `vatios()` y `empuje()`.

[16] En la práctica deberá usarse un buen programa depurador.

archivo1

```
char eleccion;
int marcador;
long fecha, tiempo;
int main( )
{
     .
     .
     .
}
double factor;
ddouble vatios ( )
{
     .
     .
     .
}
double empuje( )
{
     .
     .
     .
}
```

archivo2

```
char tipo_b;
double resistencia;
double roi( )
{
     .
     .
     .
}
double promedio( )
{
     .
     .
     .
}
double variancia( )
{
     .
     .
     .
}
```

Figura 6.20 *Archivos para el ejercicio 6.*

6.7 Errores comunes de programación

Un error de programación demasiado común relacionado con las funciones es transmitir tipos de datos incorrectos. Los valores transmitidos a una función deben corresponder a los tipos de datos de los parámetros declarados para la función. Una forma de verificar que se han recibido valores correctos es desplegar todos los valores transmitidos dentro del cuerpo de una función antes que se hagan cálculos. Una vez que ha tenido lugar esta verificación, puede prescindirse del despliegue.[16]

Otro error común puede ocurrir cuando se declara la misma variable en forma local dentro de la función que llama y la función que es invocada. Aun cuando el nombre de la variable es el mismo, un cambio a una variable local no altera el valor en la otra variable local.

Un error relacionado con esto puede ocurrir cuando una variable local tiene el mismo nombre que una variable global. Dentro de la función que la declara, el uso del nombre de la variable sólo afecta al contenido de la variable local a menos que se utilice el operador de resolución de alcance, **::**.

Otro error común es omitir el prototipo de la función llamada ya sea antes o dentro de la función que llama. La función invocada debe ser alertada del tipo de valor que se devolverá, y esta información es proporcionada por el prototipo de la función. El prototipo puede omitirse si la función llamada está colocada físicamente en un programa antes de la función que la llama. Aunque también es permisible omitir el prototipo y el tipo de devolución para funciones que devuelven un número entero, es una mala práctica de documentación hacerlo de esta manera. El valor real devuelto por una función puede verificarse desplegándolo antes y después de que es devuelto.

Los últimos dos errores comunes son terminar una línea de encabezado de una función con un punto y coma, y olvidar incluir el tipo de datos de los parámetros de una función dentro de la línea de encabezado.

6.8 RESUMEN DEL CAPÍTULO

1. Una función es invocada dando su nombre y transmitiéndole cualesquier datos que haya en el paréntesis que sigue al nombre. Si una variable es uno de los argumentos en una llamada a la función, la función llamada recibe una copia del valor de la variable.
2. La forma usada de una función escrita por el usuario por lo general es

   tipo-de-datos-a-devolver nombre-de-función(lista de parámetros)

   {

   declaraciones y otras instrucciones de C++;

   return expresion;

   }

   La primera línea de la función se llama **encabezado de la función.** Las llaves de apertura y cierre de la función y todas las instrucciones entre estas llaves constituyen el **cuerpo** de la función. El tipo de datos devuelto, por omisión, es un número entero cuando no se especifica ningún tipo de datos devuelto. La lista de parámetros es una lista de declaraciones de parámetros separadas por comas.
3. El tipo de devolución de una función es el tipo de datos del valor devuelto por la función. Si no se declara ningún tipo se asume que la función devuelve un valor entero. Si la función no devuelve un valor deberá declararse como un tipo `void`.
4. Las funciones pueden devolver en forma directa cuando mucho un valor de un solo tipo de datos a las funciones que las llaman. Este valor es el valor de la expresión en la instrucción de devolución.

5. Usando argumentos de referencia, se puede transmitir a una función la dirección de una variable. Si a una función llamada se le transmite una dirección tiene la capacidad de tener acceso directo a la variable de la función que llama respectiva. Usar direcciones transmitidas permite que una función llamada devuelva de manera efectiva múltiples valores.

6. Las funciones pueden declararse a todas las funciones que llaman por medio de un **prototipo de función.** El prototipo proporciona una declaración para una función que especifica el tipo de datos devuelto por la función, su nombre y los tipos de datos de los argumentos esperados por la función. Como con todas las declaraciones, un prototipo de función se termina con un punto y coma y puede incluirse dentro de declaraciones de variable local o como una declaración global. La forma más común de un prototipo de función es:

   tipo-de-datos nombre-de-función(lista de tipos de datos de los parámetros);

   Si la función llamada es colocada físicamente arriba de la función que llama, no se requieren más declaraciones porque la definición de la función sirve como una declaración global para todas las funciones que siguen.

7. Todas las variables usadas en un programa tienen un **alcance,** el cual determina en qué parte del programa puede utilizarse la variable. El alcance de una variable es local o global y está determinado por el lugar donde se coloca la instrucción de definición de la variable. Una variable local se define dentro de una función y sólo puede usarse dentro de la función o bloque que la define. Una variable global se define fuera de una función y puede usarse en cualquier función que siga a la definición de la variable. Todas las variables globales que no son inicializadas de manera específica por el usuario son inicializadas en cero por el compilador y pueden ser compartidas entre archivos que usen la palabra clave `extern`.

8. Todas las variables tienen una **clase.** La clase de una variable determina cuánto se conservará el valor en la variable, lo cual también se conoce como la duración de la variable: las variables `auto` son variables locales que sólo existen mientras se está ejecutando la función que las define; las variables `register` son similares a las variables `auto` pero son almacenadas en los registros internos de una computadora y no en la memoria; las variables `static` pueden ser globales o locales y conservan sus valores mientras dura la ejecución de un programa. Todas las variables `static` se colocan en cero cuando se definen si no son inicializadas de manera explícita por el usuario.

Consideración de opciones de carrera

Ingeniería química

La ingeniería química es la aplicación del conocimiento o técnicas de la ciencia, en particular de la química, a la industria. Los ingenieros químicos son responsables del diseño y operación de plantas manufactureras, a gran escala, para materiales que experimentan cambios químicos en su producción. Estos materiales incluyen todos los productos nuevos y mejorados que han afectado en forma tan profunda a la sociedad, como petroquímicos, hules y polímeros, nuevas aleaciones de metales, sustancias químicas industriales y refinados, alimentos, pinturas, detergentes, cementos, pesticidas, gases industriales y medicinas. Los ingenieros químicos también desempeñan un papel importante en el abatimiento de la contaminación y la administración de los recursos energéticos existentes. Debido a que el campo de la ingeniería química ha crecido tanto, es difícil clasificar las actividades de los ingenieros químicos. A grandes rasgos pueden dividirse en sistemas de producción a gran escala, o procesamiento químico, y en sistemas a escala más pequeña, o moleculares.

Procesamiento químico

El procesamiento químico tiene que ver con todos los aspectos del diseño y operación de plantas de procesamiento químico grandes. Incluye las siguientes áreas:

1. Petroquímicos. La destilación y refinación de combustibles como gasolina, gas natural sintético, licuefacción y gasificación del carbón, y la producción de una variedad infinita de productos hechos de petróleo, desde cosméticos hasta productos farmacéuticos.
2. Materiales sintéticos. El proceso de polimerización, la unión de moléculas más simples en moléculas complejas grandes, es responsable de muchos materiales modernos como el nailon, hules sintéticos, poliestireno, y una gran variedad de plásticos y fibras sintéticas.
3. Ingeniería de alimentos y bioquímica. La manufactura de alimentos empacados, aditivos alimenticios mejorados, esterilización y la utilización de bacterias, hongos y levaduras industriales en procesos como la fermentación.
4. Operaciones unitarias. El análisis del transporte de calor o líquidos, como el bombeo de sustancias químicas a través de una tubería o la transferencia de calor entre sustancias. Esta área incluye también el efecto de la transferencia de calor en reacciones químicas como la oxidación, la clorinación, etcétera.
5. Ingeniería criogénica. El diseño de plantas que operan a temperaturas cercanas al cero absoluto.
6. Ingeniería electroquímica. El uso de electricidad para alterar reacciones químicas, como el electrochapado, o el diseño de baterías o celdas de energía.
7. Control de la contaminación. Un campo que ha tenido un crecimiento rápido que busca vigilar y reducir los efectos perjudiciales del procesamiento químico en el ambiente. Sus áreas de interés son el control del desperdicio de agua, el abatimiento de la contaminación del aire y los aspectos económicos del control de la contaminación.

(continúa)

Consideración de opciones de carrera

Sistemas moleculares

Este campo implica la aplicación de técnicas de laboratorio en procesos en gran escala. Incluye las siguientes áreas:

1. Ingeniería bioquímica. Aplicación de enzimas, bacterias u otros microorganismos para mejorar los procesos químicos a gran escala.
2. Síntesis de polímeros. La base molecular de las propiedades de los polímeros y la síntesis química de nuevos polímeros adaptados a la producción a gran escala.
3. Investigación y desarrollo en todas las áreas del procesamiento químico.

La preparación para una carrera en la ingeniería química requiere una educación sólida en física, química y matemáticas y un conocimiento de termodinámica y química física, analítica y orgánica. Aunque con una capacitación extensa en química, los ingenieros químicos difieren de los químicos, ya que su principal interés es la adaptación de las técnicas de laboratorio a plantas manufactureras a gran escala.

CAPÍTULO 7
Completar lo básico

TEMAS

7.1 MANEJO DE EXCEPCIONES

7.2 LA CLASE `string`
FUNCIONES DE LA CLASE `string` PRECAUCIÓN: EL CARÁCTER DE NUEVA LÍNEA FANTASMA
ENTRADA Y SALIDA DE `string` PROCESAMIENTO DE CADENA

7.3 MÉTODOS DE MANIPULACIÓN DE CARACTERES
E/S DE CARACTERES
LA NUEVA LÍNEA FANTASMA REVISADA DE NUEVO
UNA SEGUNDA MIRADA A LA VALIDACIÓN DE ENTRADAS DEL USUARIO

7.4 VALIDACIÓN DE DATOS DE ENTRADA

7.5 ESPACIO DE NOMBRES Y CREACIÓN DE UNA BIBLIOTECA PERSONAL

7.6 ERRORES COMUNES DE PROGRAMACIÓN

7.7 RESUMEN DEL CAPÍTULO

El estándar ANSI/ISO actual para C++ introduce dos características nuevas que no eran parte de la especificación original de C++: el manejo de excepciones y una clase `string`. *Ambas características se presentan en este capítulo.*

El manejo de excepciones es un medio de detección y procesamiento de errores, el cual ha obtenido una aceptación creciente en la tecnología de programación. Permite detectar un error en el punto en el código en que ha ocurrido y proporciona un medio de procesarlo y regresar el control a la línea que lo generó. Aunque dicha detección de errores y corrección del código es posible usando instrucciones y funciones `if`, *el manejo de excepciones proporciona una herramienta de programación más útil dirigida a la detección y procesamiento de errores.*

Con el nuevo estándar ANSI/ISO de C++, una clase llamada `string` *es parte ahora de la biblioteca C++ estándar. Esta clase proporciona un conjunto muy amplio*

de funciones de clase que incluye la inserción y eliminación fácil de caracteres de una cadena, la expansión automática de la cadena siempre que se exceda su capacidad original, la contracción de la cadena cuando se eliminen caracteres, y comprobación del rango para detectar posiciones de caracteres inválidas.

Además de presentar estas dos características nuevas de C++, este capítulo muestra cómo el manejo de excepciones, cuando se aplica a las cadenas, proporciona un medio muy útil para validar la entrada del usuario.

7.1 MANEJO DE EXCEPCIONES

El enfoque tradicional de C++ para el manejo de errores usa una función para devolver un valor específico para indicar operaciones específicas. Por lo general, se usa un valor devuelto de 0 o 1 para indicar una ejecución exitosa de la tarea de la función, mientras que se usa un valor negativo para indicar una condición de error. Por ejemplo, si se utiliza una función para dividir dos números, podría usarse un valor devuelto de −1 para indicar que el denominador era cero y que la división no podía llevarse a cabo. Cuando pueden ocurrir múltiples condiciones de error, se usarán diferentes valores de devolución para indicar errores específicos.

Aunque este enfoque aún está disponible y se usa con frecuencia, pueden ocurrir varios problemas con este método. Primero, el programador debe verificar el valor de devolución para detectar si ocurrió un error. A continuación, el código para el manejo de error que verifica el valor devuelto con frecuencia se entremezcla con el código de procesamiento normal, dificultando a veces determinar cuál parte del código está manejando errores y el procesamiento normal del programa. Por último, devolver una condición de error de una función significa que la condición debe ser del mismo tipo de datos que un valor devuelto válido; por tanto, el código de error debe ser un valor identificado de manera especial que pueda ser reconocido como una alerta de error. Esto significa que el código de error está incrustado de manera efectiva como uno de los valores de no error posibles que pueden ser requeridas desde la función y sólo está disponible en el punto donde la función devuelve un valor. Una función que devuelve un valor booleano no tiene valores adicionales que puedan utilzarse para reportar una condición de error.

Nada de esto es insuperable y muchas veces este enfoque es simple y efectivo. Sin embargo, en las versiones más recientes, los compiladores de C++ han agregado una técnica diseñada de manera específica para la detección y manejo de errores conocida como manejo de excepciones.

En el ***manejo de excepciones,*** cuando ocurre un error mientras se está ejecutando una función, el método crea un valor, variable u objeto, el cual se conoce como ***excepción,*** que contiene información sobre el error en el punto en que ocurre. Esta excepción es transmitida de inmediato, en el punto en que fue generada, al código que se denomina ***manejador de excepciones,*** el cual está diseñado para ocuparse de la excepción. El proceso de generar y transmitir la excepción en el punto en que se detectó el error se conoce como ***lanzar una excepción.*** La excepción es lanzada desde dentro de la función mientras aún se está ejecutando. Esto permite manejar el error y luego regresar el control de nuevo a la función de modo que pueda completar su tarea.

En general, dos tipos de errores fundamentales pueden causar excepciones en C++: aquellos que resultan de una incapacidad del programa para obtener un recurso requerido, y aquellos que resultan de datos defectuosos. Son ejemplos del primer tipo de error los

intentos de obtener un recurso del sistema, como localizar y encontrar un archivo para entrada. Estos tipos de errores son el resultado de recursos externos sobre los cuales el programador no tiene control.

El segundo tipo de error puede ocurrir cuando un programa le indica al usuario que introduzca un número entero, y el usuario introduce una cadena, como e234, que no puede ser convertida en un valor numérico. Otro ejemplo es el intento de dividir dos números cuando el denominador tiene un valor de 0. Esta última condición se conoce como error por división entre cero. Cada uno de estos errores siempre puede verificarse y manejarse de una manera que no produzca una falla del programa. Antes de ver cómo se logra esto usando el manejo de excepciones, revise la tabla 7.1 para familiarizarse con la terminología usada en relación con las excepciones de procesamiento.

Tabla 7.1 Terminología del manejo de excepciones

Terminología	Descripción
Excepción	Un valor, variable u objeto que identifica un error específico que ha ocurrido mientras se está ejecutando un programa
Lanzar una excepción	Envía la excepción a una sección de código que procesa el error detectado
Atrapar o manejar una excepción	Recibe una excepción lanzada y la procesa
Cláusula catch	La sección de código que procesa el error
Manejador de excepción	El código que lanza y atrapa una excepción

La sintaxis general del código requerido para lanzar y atrapar una excepción es la siguiente:

```
try
{
  // una o mas instrucciones,
  // al menos una de las cuales debería
  // ser capaz de lanzar una excepción;
}
catch(Tipo-de-datos-de-excepción nombre-de-parámetro)
{
  // una o más instrucciones
}
```

Este ejemplo usa dos nuevas palabras clave: `try` y `catch`.

La palabra clave `try` identifica el inicio de un bloque de código para manejar una excepción. Al menos una de las instrucciones dentro de las llaves que definen a este bloque de código deberá ser capaz de lanzar una excepción. Como ejemplo, examínese el bloque `try` en la siguiente sección de código:

```
try
{
  cout << "Introduzca el numerador (sólo números enteros): ";
  cin >> numerador;
  cout << "Introduzca el denominador (sólo números enteros): ";
  cin >> denominador;
  resultado = numerador/denominador;
}
```

El bloque `try` contiene cinco instrucciones, tres de las cuales pueden producir un error que usted desea descubrir. En particular, un programa escrito de manera profesional dbería asegurar que se introducen números enteros válidos en respuesta a ambos indicadores y que el segundo valor introducido no sea un cero. Para este ejemplo, sólo comprobará que el segundo valor introducido no es un cero. (En el apéndice C encontrará el código de manejo de excepciones que puede usarse para validar ambas entradas y asegurar que ambos datos introducidos son números enteros.)

Desde el punto de vista del bloque `try`, sólo importa el valor del segundo número. El bloque `try` se alterará para expresar "pruebe todas las instrucciones para ver si ocurre una excepción, la cual en este caso particular es un segundo valor cero". Para comprobar que el segundo valor no es un cero, se agrega una instrucción `throw` dentro del bloque `try`, como sigue:

```
try
{
  cout << "Introduzca el numerador (sólo números enteros): ";
  cin >> numerador;
  cout << "Introduzca el denominador (sólo números enteros): ";
  cin >> denominador;
  if (denominador == 0)
    throw denominador;
  else
    resultado = numerador/denominador;
}
```

En este bloque `try`, el elemento lanzado es un valor entero. Podría haberse usado un literal de cadena, una variable o un objeto, pero sólo uno de estos elementos puede ser lanzado por cualquier instrucción throw individual. Las primeras cuatro instrucciones en el bloque `try` no tienen que incluirse en el código; sin embargo, hacerlo así mantiene juntas todas las instrucciones relevantes. Mantener juntas las instrucciones relacionadas puede facilitar agregar instrucciones de lanzamiento dentro del mismo bloque `try` para asegurar que ambos valores de entrada son valores enteros, así que es más conveniente tener todo el código relevante disponible dentro del mismo bloque `try`.

Un bloque `try` debe ir seguido por uno o más bloques `catch`, los cuales sirven como manejadores de excepciones para cualesquiera excepciones lanzadas por las instrucciones en el bloque `try`. Aquí hay un bloque `catch` que maneja la excepción lanzada, la cual es un número entero:

```
catch(int e)
  {
    cout << "Un valor del denominador de " << e << " es inválido." << endl;
    exit (1);
  }
```

El manejo de excepciones proporcionado por este bloque `catch` es una instrucción de salida que identifica la excepción particular descubierta y luego termina la ejecución del programa. Observe los paréntesis que siguen a la palabra clave `catch`. Dentro de los paréntesis se enlista el tipo de datos de la excepción que es lanzada y un nombre de parámetro (el cual es `e`) que se usa para recibirlo. Este identificador, el cual es seleccionado por el programador pero que usa por convención la letra `e` por excepción, se usa para mantener el valor de la excepción generado cuando se lanza una excepción.

Pueden proporcionarse múltiples bloques `catch` en tanto cada bloque atrape un tipo de datos único. El requisito es que al menos se proporcione un bloque `catch` para cada bloque `try`. Entre más excepciones puedan ser descubiertas con el mismo bloque `try`, es mejor. El programa 7.1 proporciona un programa completo que incluye un bloque `try` y un bloque `catch` para detectar un error de división entre cero.

Programa 7.1

```
#include <iostream>
using namespace std;

int main()
{
      int numerador, denominador;

  try
  {
    cout << "Introduzca el numerador (sólo números enteros): ";
    cin >> numerador;
    cout << "Introduzca el denominador (sólo números enteros): ";
    cin >> denominador;
      if (denominador == 0)
        throw denominador; // se lanza un valor entero
    else
        cout << numerador << '/' << denominador
             << " = " << double(numerador)/double(denominador) << endl;
  }
  catch(int e)
  {
    cout << "Un valor del denominador de " << e << " es inválido." << endl;
    exit (1);
  }

  return 0;
}
```

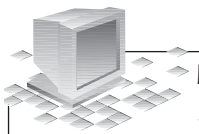
A continuación se encuentran dos muestras de ejecución usando el programa 7.1. Hay que observar que la segunda salida indica que se ha detectado con éxito un intento de dividir entre un denominador de cero antes que se realice la operación.

```
Introduzca el numerador (sólo números enteros): 12
Introduzca el denominador (sólo números enteros): 3
12/3 = 4
```

e

```
Introduzca el numerador (sólo números enteros): 12
Introduzca el denominador (sólo números enteros): 0
Un valor del denominador de 0 es inválido.
```

Habiendo detectado un denominador de cero, en lugar de terminar la ejecución del programa, un programa más robusto puede proporcionarle al usuario la oportunidad de reintroducir un valor diferente de cero. Esto puede lograrse incluyendo el bloque `try` dentro de una instrucción `while` y luego hacer que el bloque `catch` devuelva el control del programa a la instrucción `while` después de informar al usuario que se ha introducido un valor de cero. El siguiente código logra esto:



Programa 7.2

```
#include <iostream>
using namespace std;

int main()
{
  int numerador, denominador;
  bool Denominador_necesario = verdadero;

  cout << "Introduzca el numerador (sólo números enteros): ";
  cin >> numerador;

  cout << "Introduzca el denominador (sólo números enteros): ";
  while(Denominador_necesario)
  {
    cin >> denominador;
    try
    {
      if (denominador == 0)
        throw denominador;  // se lanza un valor entero
    }
    catch(int e)
    {
      cout << "Un valor del denominador de " << e << " es inválido." << endl;
      cout << "Por favor reintroduzca el denominador (solo números enteros): ";
      continue;  // esto regresa el control a la instruccion while
    }
    cout << numerador << '/' << denominador
         << " = " << double(numerador)/double(denominador) << endl;
      Denominador_necesario = falso;
  }

  return 0;
}
```

Al revisar este código, hay que observar que es la instrucción `continue` dentro del bloque `catch` la que devuelve el control a la parte superior de la instrucción `while` (véase la sección 6.3 para una revisión de la instrucción `continue`). A continuación hay una muestra de ejecución usando el programa 7.2:

```
Introduzca el numerador (sólo números enteros): 12
Introduzca el denominador (sólo números enteros): 0
Un valor del denominador de 0 es inválido.
Por favor reintroduzca el denominador (sólo números enteros): 5
12/5 = 2.4
```

Debe hacerse una advertencia cuando se lancen literales de cadena en lugar de valores numéricos. Siempre que se lance un literal de cadena, se trata de una cadena C, no un objeto de clase `string`. Esto significa que la instrucción `catch` debe declarar el argumento recibido como una cadena C, la cual es un arreglo de caracteres, en vez de una cadena. Como ejemplo, considérese que en lugar de lanzar el valor de la variable `denominador` en los programas 7.1 y 7.2, se usa la siguiente instrucción:

```
throw "***Entrada invalida – No se permite un valor del denominador de cero***";
```

Aquí se encuentra una instrucción `catch` correcta para la instrucción `throw` precedente:

```
catch(char e[])
```

Un intento de declarar la excepción como una variable de clase `string` producirá un error de compilador.

Ejercicios 7.1

1. Defina los siguientes términos:

   excepción
   bloque `try`
   bloque `catch`
   manejador de excepción
   lanzar una excepción
   atrapar una excepción

2. Introduzca y ejecute el programa 7.1.

3. Reemplace la instrucción

   ```
   cout << numerador << '/' << denominador
        << " = " << double(numerador)/double(denominador) << endl;
   ```

   en el programa 7.1 con la instrucción

   ```
   cout << numerador << '/' << denominador
        << " = " << numerador/denominador << endl;
   ```

   y ejecute el programa modificado. Introduzca los valores 12 y 5 y explique por qué es incorrecto el resultado desde el punto de vista del usuario.

4. Modifique el programa 7.1 de modo que lance y atrape el mensaje `***Entrada invalida – No se permite un valor del denominador de cero***.` (*Sugerencia:* Revise la advertencia presentada al final de esta sección.)
5. Introduzca y ejecute el programa 7.2.
6. Modifique el programa 7.2 de modo que continúe dividiendo dos números hasta que el usuario introduzca el carácter q (como numerador o denominador) para terminar la ejecución del programa.
7. Incluya el código de manejo de excepciones proporcionado en esta sección dentro del programa 7.1 para asegurar que el usuario introduce un valor entero válido tanto para el numerador como para el denominador.

7.2 LA CLASE `string`

Los programas en este texto han usado el objeto `cout` de la clase `istream` en forma extensa sin haber investigado esta clase o la manera en que se crea el objeto `cout`. Ésta es una de las ventajas del diseño de programas orientado a objetos; pueden usarse clases probadas en forma minuciosa sin conocer los detalles internos de cómo está construida la clase. En esta sección, usaremos otra clase proporcionada por la biblioteca estándar de C++, la clase `string`. Sin embargo, en este caso, se crearán objetos de la clase antes de usarlos, en lugar de usar un objeto existente, como `cout`.

Una clase es un tipo de datos creado por un usuario. Como los tipos de datos integrados, una clase define un conjunto válido de valores de datos y un conjunto de operaciones que pueden utilizarse en ellos. La diferencia entre una clase creada por un usuario y un tipo integrado es la forma en que está construida la clase. Un tipo de datos integrado es proporcionado como una parte integral del compilador, y una clase es construida por un programador usando código C++. Aparte de eso y de la terminología usada, los dos tipos se utilizan en forma muy parecida. La diferencia fundamental en la terminología es que las áreas de almacenamiento para los tipos integrados se conocen como variables, mientras las áreas de almacenamiento declaradas para una clase se conocen como objetos.

Los valores permitidos por la clase `string` se conocen como literales de cadena. Un literal de cadena es cualquier secuencia de caracteres encerrada entre comillas. Una literal de cadena también se conoce como un valor de cadena, una constante de cadena y, de manera más convencional, una cadena. Son ejemplos de cadenas `"Esta es una cadena"`, `"¡Hola mundo!"` y `"xyz 123 *!#@&"`. Las comillas indican los puntos inicial y final de la cadena y nunca se almacenan con ella.

La figura 7.1 muestra la representación de programación de la cadena `Hielo` siempre que se crea esta cadena como un objeto de la clase `string.` Por convención, al primer carácter en una cadena siempre se le asigna una posición 0. Este valor de posición también se conoce como valor índice del carácter y valor de compensación.

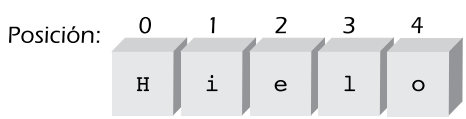


Figura 7.1 El almacenamiento de una cadena como una secuencia de caracteres.

Funciones de la clase string

La clase string proporciona diversas funciones para declarar, crear e inicializar una cadena. En las versiones anteriores de C++, el proceso de crear un objeto nuevo se conoce como instanciar un objeto, lo cual en este caso se convierte en instanciar un objeto de cadena, o crear una cadena, para abreviar. La tabla 7.2 enlista las funciones proporcionadas por la clase string para crear e inicializar un objeto de cadena. En terminología de clases, las funciones se conocen de manera formal como métodos, y los métodos que llevan a cabo esta tarea se denominan métodos constructores, o constructores, para abreviar.

Tabla 7.2 Constructores de clase string (requiere el archivo de encabezado string)

Constructor	Descripción	Ejemplos

`string nombreObjeto = valor`	Crea e inicializa un objeto de cadena a un valor que puede ser un literal de cadena, un objeto de cadena declarado con anterioridad o una expresión que contiene literales de cadena y objetos de cadena	`string str1 = "Buenos dias";`
		`string str2 = str1;`
		`string str3 = str1 + str2;`

`string nombreObjeto (valorCadena)`	Produce la misma inicialización que el anterior	`string str1 ("Hot");` `string str1 (str1 + " Dog");`
`string nombreObjeto (str, n)`	Crea e inicializa un objeto de cadena con una subcadena del objeto de cadena `str`, iniciando en la posición índice `n` de `str`	`string str1(str2, 5)` Si `str2` contiene la cadena `Buenos dias`, entonces `str1` se convierte en la cadena `dias`
`string nombreObjeto (str, n, p)`	Crea e inicializa un objeto de cadena con una subcadena del objeto de cadena `str`, iniciando en la posición índice `n` de `str` y contiene `p` caracteres	`string str1(str2, 5,2)` Si `str2` contiene la cadena `Buenos dias`, entonces `str1` se vuelve la cadena `di`
`string nombreObjeto (n, char)`	Crea e inicializa un objeto de cadena con `n` copias de `char`	`string str1(5,'*')` Esto hace a `str1 = "*****"`
`string nombreObjeto;`	Crea e inicializa un objeto de cadena para representar una secuencia de caracteres vacía (igual a la cadena `nombreObjeto = "";` el largo de la cadena es `0`)	`string mensaje;`

El programa 7.3 ilustra ejemplos de cada uno de los métodos constructores proporcionados por la clase `string`.

Programa 7.3

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1; // una cadena vacía
  string str2("Buenos dias");
  string str3 = "Hot Dog";
  string str4(str3);
  string str5(str4, 4);
  string str6 = "lineal";
  string str7(str6, 3, 3);

  cout << "str1 es: " << str1 << endl;
  cout << "str2 es: " << str2 << endl;
  cout << "str3 es: " << str3 << endl;
  cout << "str4 es: " << str4 << endl;
  cout << "str5 es: " << str5 << endl;
  cout << "str6 es: " << str6 << endl;
  cout << "str7 es: " << str7 << endl;

  return 0;
}
```

Aquí está la salida creada por el programa 7.3:

```
str1 es:
str2 es: Buenos dias
str3 es: Hot Dog
str4 es: Hot Dog
str5 es: Dog
str6 es: lineal
str7 es: eal
```

Aunque esta salida es sencilla, `str1` es una cadena vacía consistente de ningún carácter; debido a que al primer carácter en una cadena se le designa como posición cero, no uno, la posición de carácter de la `D` en la cadena `Hot Dog` se localiza en la posición cuatro, lo cual se muestra en la figura 7.2.

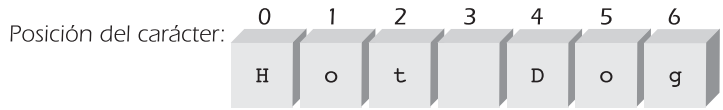


Figura 7.2 *Las posiciones de carácter de la cadena* `Hot Dog.`

Entrada y salida de `string`

Además de que una cadena se inicializa usando los métodos constructores enlistados en la tabla 7.2, las cadenas pueden ser introducidas desde el teclado y desplegarse en la pantalla. La tabla 7.3 enlista los métodos y objetos básicos que pueden usarse como entradas y salidas de valores de cadena.

Tabla 7.3 Rutinas de entrada y salida de la clase `string`

Rutina C++	Descripción
`cout`	Salida de pantalla de propósito general
`cin`	Entrada terminal de propósito general que deja de leer cuando encuentra un espacio en blanco
`getline(cin, strObj)`	Entrada terminal de propósito general que introduce todos los caracteres capturados en la cadena, `strObj`, y deja de aceptar caracteres cuando recibe un carácter de línea nueva `(\n)`

Además de los flujos `cout` y `cin` estándar, la clase `string` proporciona el método `getline()` para la entrada de cadenas. Por ejemplo, la expresión `getline(cin, mensaje)` aceptará y almacenará en forma continua caracteres mecanografiados en la terminal hasta que se oprima la tecla Entrar. Oprimir la tecla Entrar en la terminal genera un carácter de línea nueva, `'\n'`, el cual es interpretado por `getline()` como la entrada fin de línea. Todos los caracteres encontrados por `getline()`, excepto el carácter de línea nueva, son almacenados en la cadena, mensaje, como se ilustra en la figura 7.3.

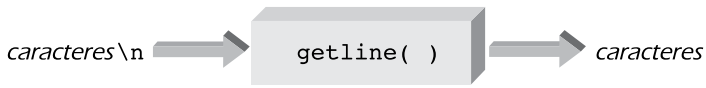


Figura 7.3 *Introducción de una cadena con* `getline()`.

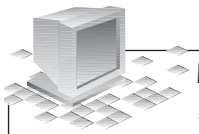
El programa 7.4 ilustra el uso del método `getline()` y el flujo `cout` para la entrada y salida de una cadena, respectivamente, que es introducida en la terminal del usuario.

La siguiente es una muestra de ejecución del programa 7.4:

```
Introduzca una cadena:
Esta es una entrada de prueba de una cadena de caracteres.
La cadena que se acaba de introducir es:
Esta es una entrada de prueba de una cadena de caracteres.
```

Aunque el objeto de flujo `cout` se usa en el programa 7.4 para la salida de cadena, por lo general el objeto de entrada de flujo `cin` no puede usarse en lugar de `getline()` para la entrada de cadena. Esto se debe a que el objeto `cin` lee un conjunto de caracteres hasta un espacio en blanco o un carácter de línea nueva. Por tanto, intentar introducir los ca-

racteres `Esta es una cadena` usando la instrucción `cin >> mensaje;` sólo produce que la palabra `Esta` se asigne a `mensaje`. El hecho que un espacio en blanco termine una operación de extracción `cin` restringe la utilidad del objeto `cin` para introducir datos de cadena y es la razón para usar `getline()`.



Programa 7.4

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string mensaje;     // declara un objeto string

  cout << "Introduzca una cadena:\n";

  getline(cin, message);

  cout << "La cadena que se acaba de introducir es:\n"
       << message << endl;


  return 0;
}
```

En su forma más general, el método `getline()` tiene la sintaxis

```
getline(cin, strObj, carácter-de-terminación)
```

donde *`strObj`* es el nombre de una variable de cadena y *`carácter-de-terminación`* es una constante, o variable, de carácter opcional que especifica el carácter de terminación. Por ejemplo, la expresión `getline(cin, mensaje, '!')` aceptará todos los caracteres introducidos en el teclado, incluyendo un carácter de línea nueva, hasta que se introduzca un signo de exclamación. El signo de exclamación no se almacenará como parte de la cadena.

Si se omite el tercer argumento opcional cuando se llama a `getline()`, el carácter de terminación por omisión es el carácter de línea nueva (`'\n'`). Por tanto, la instrucción `getline(cin, mensaje, '\n');` puede utilizarse en lugar de la instrucción `getline (cin, mensaje);`. Ambas instrucciones dejan de leer caracteres cuando se oprime la tecla Entrar. Para todos los programas usados desde este punto en adelante, se supone que la entrada es terminada al oprimir la tecla Entrar, lo cual genera un carácter de línea nueva. Así, el tercer argumento opcional transmitido a `getline()`, el cual es el carácter de terminación, se omitirá.

Precaución: El carácter de nueva línea fantasma

Al parecer pueden obtenerse resultados no deseados cuando se usan juntos el flujo de entrada `cin` y el método `getline()` para aceptar datos o cuando se utiliza el flujo de entrada `cin`, solo, para aceptar caracteres individuales. Para observar cómo puede ocurrir esto, considere el programa 7.5, el cual usa `cin` para aceptar un número entero introducido en el teclado, almacenándolo en la variable `valor`, y es seguido por una llamada al método `getline()`.

Programa 7.5

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  int valor;
  string mensaje;

  cout << "Introduzca un número: ";
  cin >> valor;
  cout << "El número introducido es:\n"
       << valor << endl;

  cout << "Introduzca texto:\n";
  getline(cin, mensaje);
  cout << "El texto introducido es:\n"
       << mensaje << endl;
  cout << int(message.length());


  return 0;
}
```

Punto de información

Los tipos de datos `string` y `char`

Una cadena puede consistir de cero, uno o más caracteres. Cuando la cadena no tiene caracteres, se dice que es una cadena vacía con una longitud de cero. Una cadena con un solo carácter, como `"a"`, es una cadena de longitud uno y se almacena de manera diferente que un tipo de datos `char`, como `'a'`. Sin embargo, para muchos propósitos prácticos, una cadena de longitud uno y un `char` responden de la misma manera; por ejemplo, `cout >> "\n"` y `cout >> '\n'` producen una línea nueva en la pantalla. Es importante entender que son tipos de datos diferentes; por ejemplo, ambas declaraciones

```
string s1 = 'a';  // INICIALIZACION INVALIDA
char key = "\n"; // INICIALIZACION INVALIDA
```

producen un error de compilador debido a que intentan inicializar un tipo de datos con valores literales de otro tipo.

Cuando se ejecuta el programa 7.5, el número introducido en respuesta al indicador `Introduzca un número:` se almacena en la variable `valor`. En este punto, todo parece funcionar bien. Hay que observar, sin embargo, que al introducir un número, se introduce un número y se oprime la tecla Entrar. En casi todos los sistemas de cómputo, estos datos introducidos se almacenan en un área de contención temporal llamada búfer inmediatamente después que se han introducido los caracteres, como se ilustra en la figura 7.4.

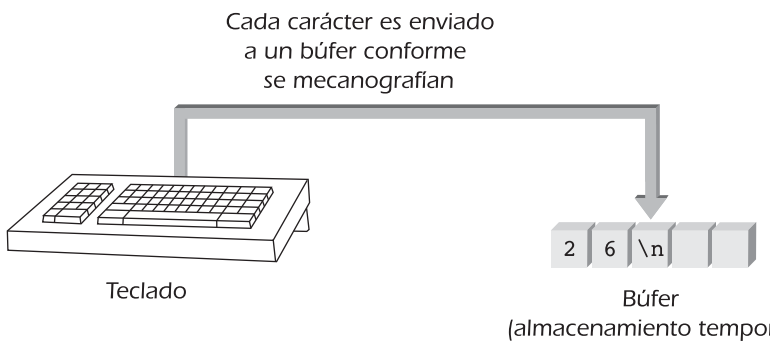


Figura 7.4 *Los caracteres mecanografiados en el teclado se almacenan primero en un búfer.*

El flujo de entrada `cin` en el programa 7.5 acepta primero el número introducido pero deja `'\n'` en el búfer. La siguiente instrucción de entrada, la cual es una llamada a `getline()`, recoge el código para la tecla Entrar como el siguiente carácter y termina cualquier entrada posterior. A continuación se presenta una muestra de ejecución para el programa 7.5:

```
Introduzca un número: 26
El número introducido es 26
Introduzca texto:
El texto introducido es
```

En esta salida, no se acepta ningún texto en respuesta al indicador `Introduzca texto:`. No ocurre ningún texto porque, después que ha sido aceptado el número 26 por el programa, el código para la tecla Entrar, la cual es una secuencia de escape de línea nueva, permanece en el búfer y es recogida e interpretada por el método `getline()` como el final de su entrada. Esto ocurrirá ya sea que sea aceptado por `cin` un número entero, como en el programa 7.5, una cadena o cualquier otra entrada y luego sea seguido por una llamada al método `getline()`.

Hay tres soluciones separadas para este problema de la tecla Entrar "fantasma":

- No mezclar entradas `cin` con `getline()` en el mismo programa.
- Hacer que a la entrada `cin` siga la llamada a `cin.ignore()`.
- Aceptar la tecla Entrar en una variable de carácter y luego ignorarla.

La solución más usada es la primera. Sin embargo, todas las soluciones se basan en el hecho que la tecla Entrar es una entrada de carácter legítima y debe reconocerse como tal. Encontrará este problema de nuevo cuando consideremos aceptar tipos de datos `char` en la siguiente sección.

Procesamiento de cadena

Las cadenas pueden manipularse usando métodos de clase `string` o el método de un carácter a la vez descrito en la siguiente sección. La tabla 7.4 enlista los métodos de clase `string` más usados. Éstos incluyen los métodos `accessor` y `mutator` más los métodos y funciones de operador que usan los operadores aritméticos y de comparación estándares.

Tabla 7.4 Los métodos de procesamiento de la clase `string` (requiere el archivo de encabezado `string`)

Método/Operación	Descripción	Ejemplo
`int length()`	Devuelve la longitud de la cadena implícita	`string.length()`
`int size()`	Igual que la anterior	`string.size()`
`at(int index)`	Devuelve el carácter en el índice especificado y lanza una excepción si el índice es inexistente	`string.at(4)`
`int compare(string)`	Compara dos cadenas; devuelve un valor negativo si la cadena implicada es menor que `str`, cero si son iguales y un valor positivo si la cadena implicada es mayor que `str`	`string1.compare(string2)`
`c_str()`	Devuelve la cadena como una cadena C terminada en `null`	`string1.c_str()`

Tabla 7.4 Los métodos de procesamiento de la clase `string` (requiere el archivo de encabezado `string`) (continuación)

Método/Operación	Descripción	Ejemplo
`bool empty`	Devuelve `verdadero` si la cadena implicada está vacía; de lo contrario, devuelve `falso`	`string1.empty()`
`erase(ind,n);`	Elimina `n` caracteres de la cadena implicada, empezando en el índice `ind`	`string1.erase(2,3)`
`erase(ind)`	Elimina todos los caracteres de la cadena implicada, empezando desde el índice `ind` hasta el final de la cadena. La longitud de la cadena restante se convierte en `ind`	`string1.erase(4)`
`int find(str)`	Devuelve el índice de la primera ocurrencia de `str` dentro del objeto implicado	`string1.find("el")`
`int find(str, ind)`	Devuelve el índice de la primera ocurrencia de `str` dentro del objeto implicado, con la búsqueda comenzando en el índice `ind`	`string1.find("el", 5)`
`int find_first_of(str, ind)`	Devuelve el índice de la primera ocurrencia de cualquier carácter en `str` dentro del objeto implicado, con la búsqueda iniciando en el índice `ind`	`string1.find_first_of("lt", 6)`
`int find_first_not_of(str, ind)`	Devuelve el índice de la primera ocurrencia de cualquier carácter que no está en `str` dentro del objeto implicado, con la búsqueda comenzando en el índice `ind`	`string1.find_first_not_of("lt ,6)`
`void insert(ind, str)`	Inserta la cadena `str` en la cadena implicada, comenzando en el índice `ind`	`string.insert(4, "ahí")`

Tabla 7.4 Los métodos de procesamiento de la clase `string` (requiere el archivo de encabezado `string`) (continuación)

Método/Operación	Descripción	Ejemplo
`void replace(ind, n, str)`	Elimina `n` caracteres en el objeto implicado, comenzando en la posición del índice `ind` e insertando la cadena `str` en la posición del índice `ind`	`string1.replace(2,4,"bien")`
`string substr(ind,n)`	Devuelve una cadena consistente de `n` caracteres extraídos de la cadena implicada empezando en el índice `ind`. Si `n` es mayor que el número restante de caracteres, se usa el resto de la cadena implicada	`string2 = string1.substr(0,10)`
`void swap(str)`	Intercambia caracteres en `str` con el objeto implicado	`string1.swap(string2)`
`[ind]`	Devuelve el carácter en el índice `x`, sin verificar si `ind` es un índice válido	`string1[5]`
`=`	Asignación (también convierte una cadena C en una cadena)	`string1 = string`
`+`	Concatena dos cadenas	`string1 + string2`
`+=`	Concatenación y asignación	`string2 += string1`

`==  !=`
`<  <=`
`>  >=`
	Operadores relacionales. Devuelve `verdadero` si la relación se satisface; de lo contrario devuelve `falso`	`string1 == string2`
		`string1 <= string2`
		`string1 > string2`

El método más usado en la tabla 7.4 es el método `length()`. Éste devuelve el número de caracteres en la cadena, lo cual se conoce como la longitud de la cadena. Por ejemplo, el valor devuelto por la llamada al método `"¡Hola mundo!".length()` es 12. Como siempre, las comillas que encierran a un valor de cadena no se consideran parte de ésta. Del mismo modo, si la cadena referenciada por string1 contiene el valor `"Ten un buen dia."`, el valor devuelto por la llamada `string1.length()` es 16.

Puede compararse la igualdad de dos expresiones de cadena usando los operadores relacionales estándar. Cada carácter en una cadena es almacenado en binario usando el código ASCII o UNICODE. Aunque estos códigos son diferentes, tienen algunas características en común. En cada uno de ellos, un blanco precede (es menor que) todas las letras y números; las letras del alfabeto se almacenan en orden de la A a la Z; y los dígitos se almacenan en orden del 0 al 9. En ambos códigos de caracteres, los dígitos van antes (es decir, son menores que) los caracteres en mayúsculas, los cuales son seguidos por los caracteres en minúsculas. Por tanto, los caracteres en mayúsculas son matemáticamente menores que los caracteres en minúsculas. Cuando se comparan dos cadenas, sus caracteres individuales se comparan un par a la vez (ambos caracteres primeros, luego ambos caracteres segundos, etc.). Si no se encuentran diferencias, las cadenas son iguales; si se encuentra una diferencia, la cadena con el primer carácter en minúscula se considera la cadena más pequeña.

- "Hola" es mayor que "Adiós" porque la primera 'H' en Hola es mayor que la primera 'A' en Adiós.
- "Hola" es menor que "hola" porque la primera 'H' en Hola es menor que la primera 'h' en hola.
- "SUÁREZ" es mayor que "JÍMENEZ" porque la primera 'S' en SUÁREZ es mayor que la primera 'J' en JÍMENEZ.
- "123" es mayor que "1227" porque el tercer carácter, '3', en 123 es mayor que el tercer carácter, '2', en 1227.
- "Bejuco" es mayor que "Bebé" porque el tercer carácter, 'j', en Bejuco es mayor que el tercer carácter, 'b', en Bebé.

El programa 7.6 usa `length()` y varias expresiones relacionales dentro del contexto de un programa completo.

Programa 7.6

```
#include <iostream>

#include <string>
using namespace std;

int main()
{
  string string1 = "Hielo";
  string string2 = "Hielo polar";

  cout << "string1 es la cadena: " << string1 << endl;
  cout << "El número de caracteres en string1 es " << int(string1.length())
       << endl << endl;

  cout << "string2 es la cadena: " << string2 << endl;
  cout << "El número de caracteres en string2 es: " << int(string2.length())
       << endl << endl;

  if (string1 < string2)
    cout << string1 << " es menor que " << string2 << endl << endl;
  else if (string1 == string2)
    cout << string1 << " es igual a " << string2 << endl << endl;
  else
    cout << string1 << " es mayor que " << string2 << endl << endl;

  string1 = string1 + " polar helado";
  cout << "Después de la concatenación, string1 contiene los caracteres: " << string1 << endl;
  cout << "La longitud de esta cadena es " << int(string1.length()) << endl;

  return 0;
}
```

A continuación hay una muestra de la salida producida por el programa 7.6:

```
string1 es la cadena: Hielo
El numero de caracteres en string1 es 5
```

```
string2 es la cadena: Hielo polar
El número de caracteres en string2 es: 11

Hielo es menor que Hielo polar

Después de la concatenación, string1 contiene los caracteres: Hielo po-
lar helado
La longitud de esta cadena es 18
```

Cuando revise esta salida, refiérase a la figura 7.5, la cual muestra cómo se almacenan en la memoria los caracteres en `string1` y `string2.` La longitud de cada cadena se refiere al número total de caracteres en la cadena, y el primer carácter en cada cadena se localiza en la posición índice 0. Por tanto, la longitud de una cadena siempre es uno más que el número índice de la última posición del carácter en la cadena.

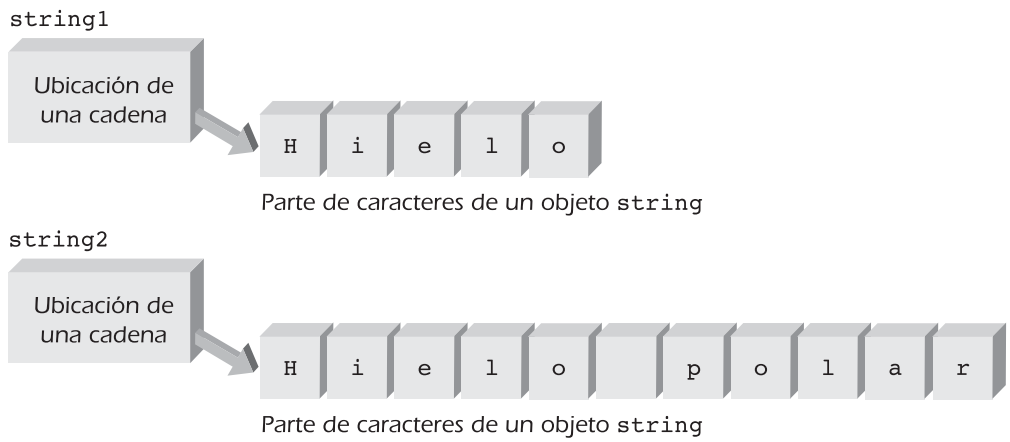
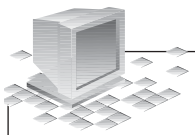


Figura 7.5 *Las cadenas iniciales usadas en el programa 7.6.*

Aunque usará fundamentalmante el operador de concatenación y el método `length()`, habrá ocasiones en que encontrará útiles los otros métodos de cadena, los cuales se describen en la tabla 7.4. Uno de los más útiles de éstos es el método `at()`, el cual le permite recuperar caracteres individuales en una cadena. El programa 7.7 utiliza este método para seleccionar un carácter a la vez de la cadena, comenzando en la posición cero de la cadena y terminando en el índice del último carácter en ella. Este último valor índice siempre es uno menos que el número de caracteres en la cadena (es decir, la longitud de la cadena).



Programa 7.7

```
#include <iostream>
#include <string>
using namespace std;

int main()
{

  string str = "Contar el número de vocales";
  int i, numChars;
  int cuentaVocales = 0;

  cout << "La cadena: " << str << endl;

  numChars = int(str.length());
  for (i = 0; i < numChars; i++)
  {
    sswitch(str.at(i)    // aqui es donde se recupera un caracter
    {
      caso 'a':
      caso 'e':
      caso 'i':
      caso 'o':
      caso 'u':
        cuentaVocales++;
    }
  }
  cout << "tiene " << cuentaVocales << " vocales." << endl;


  return 0;
}
```

La expresión `str.at(i)` en la instrucción `switch` anterior recupera el carácter en la posición `i` en la cadena. Este carácter se compara luego con cinco valores de carácter diferentes. La instrucción `switch` utiliza el hecho que los casos seleccionados son "ejecutados" en ausencia de instrucciones break. Por tanto, todos los casos seleccionados producen un incremento en `cuentaVocales.` La salida desplegada por el programa 7.7 es la siguiente:

```
La cadena: Contar el número de vocales
tiene 9 vocales.
```

Como ejemplo para insertar y reemplazar caracteres en una cadena, usando los métodos enlistados en la tabla 7.4, suponga que comienza con una cadena creada por la siguiente instrucción:

```
string str = "Debe pensar si";
```

La figura 7.6 ilustra cómo se almacena esta cadena en el búfer creado para ella. Como se indica, la longitud inicial de la cadena es de 14 caracteres.

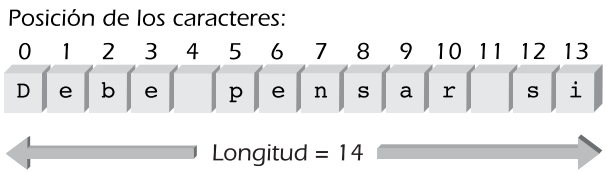


Figura 7.6 *Almacenamiento inicial de un objeto de cadena.*

Ahora suponga que se ejecuta la siguiente instrucción:

```
str.insert(4, " sentir");
```

Esta instrucción causa que se inserten los siete caracteres designados, comenzando con un espacio en blanco, a partir de la posición índice 4, en la cadena existente. La cadena resultante, después de la inserción, es como se muestra en la figura 7.7.

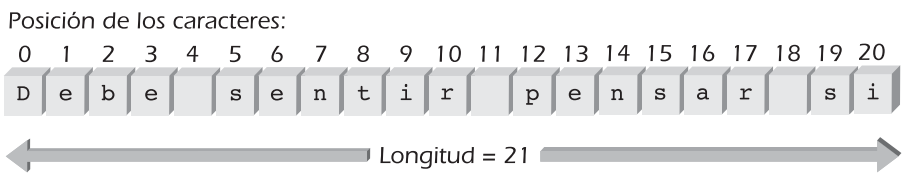


Figura 7.7 *La cadena después de la inserción.*

Si ahora se ejecuta la instrucción `str.replace(12, 6, "tu");`, los caracteres existentes en las posiciones de índice 12 a 17 se eliminarán y los dos caracteres `tu` se insertarán empezando en la posición de índice 12. Por tanto, el efecto neto del reemplazo es como se muestra en la figura 7.8. El número de caracteres de reemplazo, que en este caso particular son dos, puede ser menor que, igual a o mayor que los caracteres que se están reemplazando, los cuales en este caso son seis.

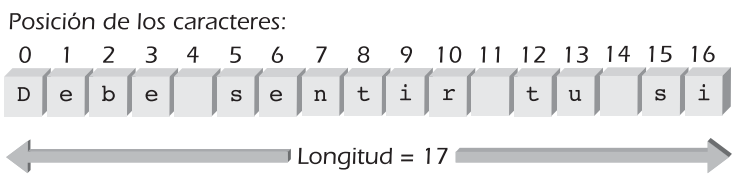


Figura 7.8 *La cadena después del reemplazo.*

Por último, si se anexa la cadena `"podrias"` a la cadena mostrada en la figura 7.8 usando el operador de concatenación, `+`, se obtiene la cadena ilustrada en la figura 7.9.

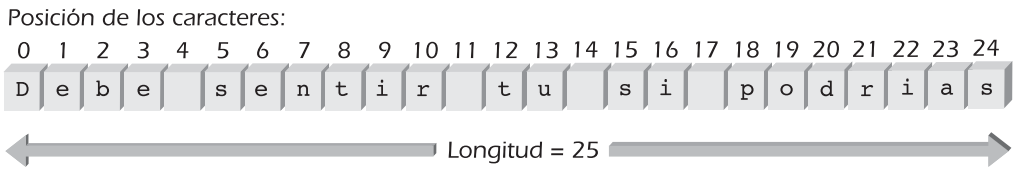


Figura 7.9 *La cadena después de la anexión.*

El programa 7.8 ilustra el uso de las instrucciones dentro del contexto de un programa completo.

Programa 7.8

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str = "Debe pensar si";

  cout << "La cadena original es: " << str << endl
       << " y tiene " << int(str.length()) << " caracteres." << endl;

  // insertar caracteres
  str.insert(4, " sentir");
  cout << "La cadena, después de la inserción, es: " << str << endl
       << " y tiene " << int(str.length()) << " caracteres." << endl;

  // reemplazar caracteres
  str.replace(12, 6, "tu");
  cout << "La cadena, después del reemplazo, es: " << str << endl
       << " y tiene " << int(str.length()) << " caracteres." << endl;

  // anexar caracteres
  str = str + " podrias";
  cout << "La cadena, después de la anexión, es: " << str << endl
       << " y tiene " << int(str.length()) << " caracteres." << endl;

   return 0;
}
```

La siguiente salida producida por el programa 7.8 corresponde a las cadenas mostradas en las figuras 7.6 a 7.9:

```
La cadena original es: Debe pensar si
  y tiene 14 caracteres.
La cadena, después de la insercion, es: Debe sentir pensar si
  y tiene 21 caracteres.
La cadena, después del reemplazo, es: Debe sentir tu si
  y tiene 17 caracteres.
La cadena, después de la anexion, es: Debe sentir tu si
podrias
  y tiene 25 caracteres.
```

De los métodos de cadena restantes enlistados en la tabla 7.4, los más usados son aquellos que localizan caracteres específicos en una cadena y crean subcadenas. El programa 7.9 presenta ejemplos de la forma en que se utilizan algunos de estos métodos.

Programa 7.9

```
#include <iostream>
#include <string>
using namespace std;

int main()
{

  string string1 = "LINEAR PROGRAMMING THEORY";
  string s1, s2, s3;
  int j, k;

  cout << "La cadena original es " << string1 << endl;

  j = int(string1.find('I'));
  cout << " La primera posición de una 'I' es " << j << endl;

  k = int(string1.find('I', (j+1)));
  cout << " La siguiente posición de una 'I' es " << k << endl;

  j = int(string1.find("THEORY"));
  cout << " La primera ubicación de \"THEORY\" es " << j << end;

  k = int(string1.find("ING"));
  cout << " El primer índice de \"ING\" es " << k << endl;
```

(Continúa)

(Continuación)

```
    s1 = string1.substr(2,5);
    s2 = string1.substr(19,3);
    s3 = string1.substr(6,8);

    cout << s1 + s2 + s3 << endl;


    return 0;
}
```

Aquí está la salida producida por el programa 7.9:

```
La cadena original es LINEAR PROGRAMMING THEORY
  La primera posicion de una 'I' es 1
  La siguiente posicion de una 'I' es 15
  La primera ubicacion de "THEORY" es 19
  El primer indice de "ING" es 15
NEAR THE PROGRAM
```

El punto principal ilustrado en el programa 7.9 es que pueden localizarse y extraerse de una cadena caracteres individuales y secuencias de caracteres.

Ejercicios 7.2

1. Introduzca y ejecute el programa 7.4.
2. Determine el valor de `text.at(0)`, `text.at(3)` y `text.at(10)`, asumiendo que el texto es, individualmente, cada una de las siguientes cadenas:
   a. ahora es el momento
   b. el mapache rocky le da la bienvenida
   c. Felices fiestas
   d. El buen barco
3. Introduzca y ejecute el programa 7.7.
4. Modifique el programa 7.7 para contar y desplegar los números individuales de cada vocal contenida en la cadena.
5. Modifique el programa 7.7 para desplegar el número de vocales en una cadena introducida por el usuario.
6. Usando el método `at()`, escriba un programa en C++ que lea en una cadena usando `getline()` y luego despliegue la cadena en orden inverso. (*Sugerencia:* Una vez que se ha introducido y guardado la cadena, recupere y despliegue caracteres empezando por el final de la cadena.)
7. Escriba un programa en C++ que acepte tanto una cadena como un solo carácter del usuario. El programa deberá determinar cuántas veces está contenido el carácter en la cadena. (*Sugerencia:* Busque la cadena usando el método `find(str,`

ind). Este método deberá utilizarse en un ciclo que comience el valor índice en cero y luego lo cambie al valor uno pasado de donde se encontró por última vez el carácter.)

8. Introduzca y ejecute el programa 7.8.
9. Introduzca y ejecute el programa 7.9.
10. Escriba un programa en C++ que acepte una cadena del usuario y luego reemplace todas las ocurrencias de la letra e con la letra x.
11. Modifique el programa escrito para el ejercicio 10 para buscar la primera ocurrencia de una secuencia de caracteres introducida por el usuario y reemplazar esta secuencia, cuando se encuentre en la cadena, con un segundo conjunto de una secuencia introducida por el usuario. Por ejemplo, si la cadena introducida es La `figura 4-4 ilustra la salida del programa 4-2` y el usuario introduce que 4- sea reemplazado por 3−, la cadena resultante será La `figura 3-4 ilustra la salida del programa 4-2`. (Sólo se ha cambiado la primera ocurrencia de lo buscado en la secuencia.)
12. Modifique el programa escrito para el ejercicio 11 para reemplazar todas las ocurrencias de la secuencia de caracteres designada con la nueva secuencia de caracteres. Por ejemplo, si la cadena introducida es La `figura 4-4 ilustra la salida del programa 4-2` y el usuario introduce que 4− ha de ser reemplazado por 3−, la cadena resultante será La `figura 3-4 ilustra la salida del programa 3-2.`

7.3 MÉTODOS DE MANIPULACIÓN DE CARACTERES

Además de los métodos `string` proporcionados por la clase string, el lenguaje C++ proporciona diversas funciones útiles de la clase `character.` Estas funciones se muestran en la tabla 7.5. Las declaraciones de función (prototipos) para cada una de estas rutinas están contenidas en el archivo de encabezado `string` y `cctype`, los cuales deben incluirse en cualquier programa que use estas funciones.

Tabla 7.5 Funciones de biblioteca de carácter (requieren el archivo de encabezado string o `cctype`)

Prototipo de la función	Descripción	Ejemplo
`int isalpha(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa una letra; de lo contrario, devuelve `falso` (número entero cero)	`isalpha('a')`

`int isalnum(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa una letra o un dígito; de lo contrario, devuelve `falso` (número entero cero)	`char key;`
		`cin >> key;`
		`isalnum(key);`

Tabla 7.5 Funciones de biblioteca de carácter (requieren el archivo de encabezado string o `cctype`) (continuación)

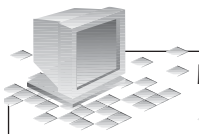
Prototipo de la función	Descripción	Ejemplo
`int isupper(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa una letra mayúscula; de lo contrario, devuelve `falso` (número entero cero)	`isupper('a')`
`int islower(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa una letra minúscula; de lo contrario, devuelve `falso` (número entero cero)	`islower('a')`
`int isdigit(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un dígito (0 a 9); de lo contrario, devuelve `falso` (número entero cero)	`isdigit('a')`
`int isascii(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un carácter ASCII; de lo contrario, devuelve `falso` (número entero cero)	`isascii('a')`
`int isspace(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un espacio; de lo contrario, devuelve `falso` (número entero cero)	`isspace(' ')`
`int isprint(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un carácter imprimible; de lo contrario, devuelve `falso` (número entero cero)	`isprint('a')`
`int isctrl(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un carácter de control; de lo contrario, devuelve `falso` (número entero cero)	`isctrl('a')`
`int ispucnt(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un carácter de puntuación; de lo contrario, devuelve `falso` (número entero cero)	`ispucnt('!')`
`int isgraph(charExp)`	Devuelve `verdadero` (número entero diferente de cero) si `charExp` evalúa un carácter imprimible diferente a un espacio en blanco; de lo contrario, devuelve `falso` (número entero cero)	`isgraph(' ')`
`int toupper(charExp)`	Devuelve el equivalente en mayúscula si `charExp` evalúa un carácter en minúscula; de lo contrario, devuelve el código de carácter sin modificación	`toupper('a')`
`int tolower(charExp)`	Devuelve el equivalente en minúscula si `charExp` evalúa un carácter en mayúscula; de lo contrario, devuelve el código de carácter sin modificación	`tolower('A')`

Debido a que todas las funciones `istype()` enlistadas en la tabla 7.5 devuelven un número entero diferente de cero (el cual es interpretado como un valor booleano verdadero) cuando el carácter satisface la condición deseada y un número entero cero (o valor boo-

leano `falso`) cuando la condición no se cumple, por lo general estas funciones se usan en forma directa dentro de una instrucción `if`. Por ejemplo, considérese el siguiente segmento de código, el cual supone que `ch` es una variable de carácter:

```
if(isdigit(ch))
  cout << "El carácter que se acaba de introducir es un dígito" << endl;
else if(ispunct(ch))
  cout << "El carácter que se acaba de introducir es un signo de pun-
  tuación" << endl;
```

En este ejemplo, si `ch` contiene un carácter de dígito, se ejecuta la primera instrucción `cout`; si el carácter es una letra, se ejecuta la segunda instrucción `cout`. En ambos casos, sin embargo, el carácter que se va a comprobar es incluido como un argumento para el método apropiado. El programa 7.10 ilustra este tipo de código dentro de un programa que cuenta el número de letras, dígitos y otros caracteres en una cadena. Los caracteres individuales que se van a comprobar se obtienen usando el método `at()` de la clase `string`. En el programa 7.10, este método se usa en un ciclo `for` que circula por la cadena desde el primer carácter hasta el último.



Programa 7.10

```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main()
{
    sstring str = "Esta -123/ es 567 A ?<6245> foca!";
    char siguienteCar;
    int i;
    int numLetras = 0, numDigitos = 0, numOtros = 0;

    cout << "La cadena original es: " << str
         << "\nEsta cadena contiene " << int(str.length())
            << " caracteres," << " la cual consiste de" << endl;

    // verifica cada carácter en la cadena
    for (i = 0; i < int(str.length()); i++)
    {
      siguienteCar = str.at(i); // obtiene un carácter
      if (isalpha(siguienteCar))
        numLetras++;
      else if (isdigit(siguienteCar))
        numDigitos++;
```

(Continúa)

(Continuación)

```
        else
          numOtros++;
      }

      cout << "      " <<  numLetras << " letras" << endl;
      cout << "      " <<  numDigitos << " digitos" << endl;
      cout << "      " <<  numOtros << " otros caracteres." << endl;

    cin.ignore();
    return 0;
}
```

La salida producida por el programa 7.10 es la siguiente:

```
La cadena original es: Esta -123/ es 567 A ?<6245> foca!
Esta cadena contiene 33 caracteres, los cuales consisten de
     11 letras
     10 digítos
     12 otros caracteres.
```

Como lo indica esta salida, cada uno de los 33 caracteres en la cadena ha sido clasificado en forma correcta como una letra, dígito u otro carácter.

Por lo general, como en el programa 7.10, cada una de las funciones en la tabla 7.5 se utiliza carácter por carácter en cada carácter en una cadena. Esto se ilustra de nuevo en el programa 7.11, donde cada carácter en minúsculas en la cadena es convertido a su equivalente en mayúsculas usando la función `toupper()`. Esta función sólo convierte las letras minúsculas, dejando todos los demás caracteres intactos.

Una muestra de la ejecución del programa 7.11 produjo la siguiente salida:

```
Mecanografíe cualquier secuencia de caracteres: esta es una prueba DE 12345.
Los caracteres recién introducidos, en mayúsculas, son: ESTA ES UNA PRUE-
BA DE 12345.
```

En el programa 7.11, ponga particular atención a la instrucción `for (i = 0; i < int(str.length()); i++)` que se utilizo para hacer un ciclo a través de cada uno de los caracteres en la cadena. Ésta es la forma típica de acceso a cada elemento en una cadena, usando el método `length()` para determinar cuando se ha alcanzado el final de la cadena. (Revise el programa 7.10 para verificar que se usa en la misma forma.) La única diferencia real es que en el programa 7.11 se tiene acceso a cada elemento usando la notación en subíndice `str[i];` en el programa 7.10 se usó el método `at()`. Aunque estas dos notaciones son intercambiables, y es cuestión de elección cuál utilizan, las dos notaciones no deberán mezclarse en el mismo programa para mantener la consistencia.

Programa 7.11

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  int i;
  string str;

  cout << "Mecanografíe cualquier secuencia de caracteres: ";
  getline(cin,str);

  // hace un ciclo a través de todos los elementos de la cadena
  for (i = 0; i < int(str.length()); i++)
    str[i] = toupper(str[i]);

  cout << "Los caracteres recién introducidos, en mayúsculas, son: "
       << str << endl;

  cin.ignore();
  return 0;
}
```

E/S de caracteres

Aunque se ha utilizado `cin` y `getline()` para aceptar los datos introducidos desde el teclado de una manera más o menos parecida a un "recetario de cocina", es necesario entender qué datos se están enviando al programa y cómo debe reaccionar éste para procesar los datos. En un nivel básico, toda la entrada (al igual que la salida) se hace carácter por carácter, como se ilustra en la figura 7.10.

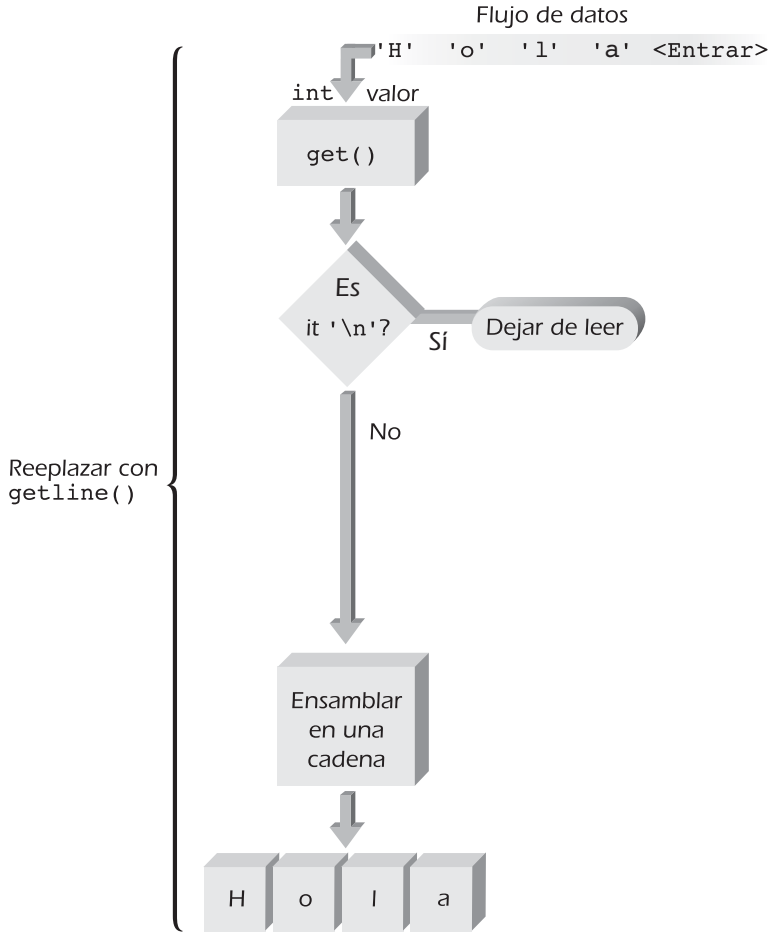


Figura 7.10 *Aceptar caracteres introducidos en el teclado.*

Como se ilustra en la figura 7.10, la introducción de cada pieza de datos, sea una cadena o un número, consiste en la mecanografía de caracteres individuales. Por ejemplo, la entrada de la cadena Hola consiste en oprimir y soltar las cuatro teclas `H`, `o`, `l`, `a` y la tecla Entrar. Del mismo modo, la salida del número `26.95` consiste en el despliegue de los cinco caracteres `2`, `6`, `.`, `9` y `5`. Aunque el programador por lo general no piensa en los datos de esta manera, el programa está restringido a esta E/S carácter por carácter, y todos los métodos y flujos E/S de nivel superior en C++ se basan en métodos de E/S de carácter de nivel inferior. Estos métodos de carácter más elemental, los cuales pueden ser usados en forma directa por un programador, se muestran en la tabla 7.6.

Punto de información

Por qué el tipo de datos `char` usa valores en números enteros

En C++, un carácter se almacena como un valor en número entero, lo cual en ocasiones es confuso para los programadores principiantes. La razón para esto es que, además de las letras y caracteres estándar en inglés, un programa necesita almacenar caracteres especiales que no tienen equivalentes imprimibles. Uno de éstos es el centinela de fin de archivo que utilizan todos los sistemas de cómputo para designar el final de un archivo de datos. Estos centinelas de fin de archivo pueden ser transmitidos desde el teclado. Por ejemplo, en sistemas basados en Unix, se genera oprimiendo el botón Ctrl y, mientras se mantiene oprimido, se oprime la tecla D; en sistemas basados en Windows, se genera oprimiendo Ctrl y, mientras se mantiene oprimido, oprimiendo la tecla Z. Ambos centinelas son almacenados como el número entero -1, el cual no tiene un valor de carácter equivalente. (Puede verificar esto desplegando el valor entero de cada carácter introducido [véase el programa 7.12] y mecanografiando la combinación Ctrl + D o la combinación Ctrl + Z, dependiendo del sistema que use.)

Al usar un valor entero de 16 bits, pueden representarse más de 64 000 caracteres diferentes. Esto proporciona suficiente almacenamiento para múltiples conjuntos de caracteres que pueden incluir árabe, chino, hebreo, japonés, ruso y casi todos los símbolos lingüísticos conocidos. Por tanto, almacenar un carácter como un valor entero tiene un valor práctico.

Una consecuencia importante de usar códigos en número entero para cadenas de caracteres es que los caracteres pueden compararse con facilidad para su ordenamiento alfabético. Por ejemplo, en tanto cada letra subsiguiente en un alfabeto tenga un valor superior que su letra precedente, la comparación de los valores de carácter se reduce a la comparación de valores numéricos. Si los caracteres se almacenan en orden numérico secuencial, esto asegura que agregar uno a una letra producirá la siguiente letra en el alfabeto.

Tabla 7.6 Métodos de E/S de carácter básicos (requieren el archivo de encabezado `cctype`)

Método	Descripción	Ejemplo
`cout.put(charExp)`	Coloca el valor de carácter de `charExp` en el flujo de salida	`cout.put('A');`
`cin.get(charVar)`	Extrae el siguiente carácter del flujo de entrada y lo asigna a la variable `charVar`	`cin.get(key);`
`cin.peek(charVar)`	Asigna el siguiente carácter del flujo de entrada a la variable `charVar` *sin extraer* el carácter del flujo	`cin.peek(nextKey);`
`cin.putback(charExp)`	Pone un valor de carácter de `charExp` de vuelta en el flujo de entrada	`cin.putback(cKey);`
`cin.ignore(n, char)`	Ignora un máximo de los siguientes `n` caracteres de entrada, hasta la detección de `char` e incluyéndola. Si no se especifican argumentos, ignora el siguiente carácter en el flujo de entrada	`cin.ignore(80,'\n');` `cin.ignore();`

Un poco de antecedentes

Una inconsistencia notacional

Todos los métodos de clase de carácter mostrados en la tabla 7.6 utilizan la notación orientada a objetos estándar en la cual el nombre del método va precedido por el nombre de un objeto, como en `cin.get()`. Éste no es el caso con el método `getline()` de la clase string el cual usa la notación `getline(cin,fstrVar)`. En esta notación, el objeto, `cin` por ejemplo, aparece como un argumento, que es la forma en la cual las funciones basadas en procedimientos transmiten las variables. En cuanto a lograr consistencia, puede esperar que `getline()` sea llamado como `cin.getline()`.

Por desgracia, la notación apropiada está en uso para un método `getline()` creado originalmente para cadenas estilo C (véase la sección 10.1); por consiguiente, fue creada una inconsistencia notacional.

La función `get()` lee el siguiente carácter en el flujo de entrada y lo asigna a la variable de carácter de la función. Por ejemplo, examine la siguiente instrucción:

```
cin.get(siguientetCar);
```

Causa que el siguiente carácter introducido en el teclado sea almacenado en la variable de carácter `siguienteCar.` Esta función es útil para introducir y verificar caracteres individuales antes que sean asignados a una cadena o tipo de datos de C++ completos.

La función de salida de carácter correspondiente a `get()` es `put()`. Esta función espera un argumento de un solo carácter y despliega el carácter que le es transmitido en la terminal. Por ejemplo, la instrucción `cout.put('A')` causa que la letra A sea desplegada en la pantalla.

De las tres últimas funciones mostradas en la tabla 7.6, la función `cin.ignore()` es la más útil. Esta función permite omitir la entrada hasta que se encuentra un carácter designado, como `'\n'`. Por ejemplo, la instrucción `cin.ignore(80, '\n')` omitirá hasta un máximo de los siguientes 80 caracteres o dejará de omitir si encuentra el carácter de línea nueva. Esta instrucción puede ser útil para omitir toda la entrada adicional en una línea, hasta un máximo de 80 caracteres, o hasta que se encuentre el final de la línea actual. La entrada comenzará con la linea siguiente.

La función `peek()` devuelve el siguiente carácter en el flujo pero no lo elimina del búfer del flujo (véase la figura 7.6). Por ejemplo, la expresión `cin.peek(siguienteCar)` devuelve el siguiente carácter introducido por medio del teclado pero lo deja en el búfer. Esto en ocasiones es útil para echar un vistazo adelante y ver cuál es el siguiente carácter, mientras se deja en su lugar para la siguiente entrada.

Por último, la función `putback()` coloca un carácter de vuelta en el flujo, de modo que será el siguiente carácter que sea leído. El argumento transmitido a `putback()` puede ser cualquier expresión de carácter que evalúe un valor de carácter legítimo y no necesite ser el último carácter introducido.

La nueva línea fantasma revisada de nuevo

Como se vio en la sección anterior, a veces se obtienen resultados aparentemente extraños cuando una entrada de flujo `cin` es seguida por una llamada al método `getline()`. Este mismo resultado puede ocurrir cuando se introducen caracteres utilizando el método de carácter `get()`. Para ver cómo puede ocurrir esto, considérese el programa 7.12, el cual usa el método `get()` para aceptar el siguiente carácter introducido en el teclado y almacenar el carácter en la variable `fkey`.

Programa 7.12

```
#include <iostream>
using namespace std;

int main()
{
  char primera_tecla;

  cout << "Mecanografie un caracter: ";
  cin.get(primera_tecla);
  cout << "La tecla que se acaba de aceptar es " << int(primera_tecla) << endl;

  return 0;
}
```

Cuando se ejecuta el programa 7.12, el carácter introducido en respuesta al indicador `Mecanografie un caracter:` se almacena en la variable de carácter `primera_tecla` y el código decimal para el carácter es desplegado al vaciar en forma explícita el carácter en un número entero, para forzar su despliegue como un valor entero. La siguiente muestra de ejecución ilustra esto:

```
Mecanografie un caracter: m
La tecla que se acaba de aceptar es 109
```

En este punto, todo parece funcionar, aunque podría preguntarse por qué se despliega el valor decimal de `m` en lugar del carácter en sí. La razón para esto se hará evidente.

Al mecanografiar `m`, por lo general se oprimen dos teclas, la tecla `m` y la tecla Entrar. Como en la sección anterior, estos dos caracteres se almacenan en un búfer después que se han oprimido (véase la figura 7.4).

La primera tecla oprimida, `m` en este caso, es tomada del búfer y almacenada en `primera_tecla`. Esto, sin embargo, aún deja el código para la tecla Entrar en el búfer. Por tanto, una llamada subsiguiente a `get()` para una entrada de carácter tomará de manera automática el código para la tecla Entrar como el siguiente carácter. Por ejemplo, considérese el programa 7.13.

Programa 7.13

```
#include <iostream>
using namespace std;

int main()
{
  char primera_tecla, segunda_tecla;

  cout << "Mecanografie un caracter: ";
  cin.get(primera_tecla);
  cout << "La tecla que se acaba de aceptar es " << int(primera_tecla) << endl;

  cout << "Mecanografie otro caracter: ";
  cin.get(segunda_tecla);
  cout << "La tecla que se acaba de aceptar es " << int(segunda_tecla) << endl;

  return 0;
}
```

La siguiente es una muestra de la ejecución del programa 7.13.

```
Mecanografie un caracter: m
La tecla que se acaba de aceptar es 109
Mecanografie otro caracter: La tecla que se acaba de acep-
tar es 10
```

Después de introducir la letra `m` en respuesta al primer indicador, también se oprime la tecla Entrar. Desde un punto de vista de carácter, esto representa la introducción de dos caracteres distintos. El primer carácter es `m`, el cual es codificado y almacenado como el número entero 109. El segundo carácter también se almacena en el búfer con el código numérico para la tecla Entrar. La segunda llamada a `get()` toma este código de inmediato, sin esperar que se oprima cualquier tecla adicional. El último flujo `cout` despliega el código para esta tecla. La razón para desplegar el código numérico en lugar del carácter en sí se debe a que la tecla Entrar no tiene un carácter imprimible asociado con ella que pueda ser desplegado.

Recuerde que cada tecla tiene un código numérico, incluyendo las teclas Entrar, Escape y Control, así como la barra espaciadora. Estas teclas por lo general no tienen efecto cuando se introducen números debido a que los métodos de entrada las ignoran como entradas a la izquierda o a la derecha de los datos numéricos. Tampoco afectan la introducción de un solo carácter solicitado como el primer dato del usuario que se va a introducir, como en el caso del programa 7.12. Sólo cuando se solicita un carácter después que el usuario ya ha introducido algún otro dato, como en el programa 7.13, se hace evidente la tecla Entrar por lo general invisible.

En la sección 7.2, aprendió algunas formas para prevenir que la tecla Entrar sea aceptada como una entrada de carácter legítima cuando se usa el método `getline()`. Las siguientes formas pueden emplearse cuando se utiliza el método `get()` dentro de un programa:

- Hacer que la entrada `cin.get()` sea seguida de la llamada `cin.ignore()`.
- Aceptar la tecla Entrar en una variable de carácter, y luego no usarla más.

El programa 7.14 aplica la primera solución al programa 7.13. Ignorar la tecla Entrar después que se ha leído y desplegado el primer carácter vacía el búfer de la tecla Entrar y lo deja listo para almacenar el siguiente carácter válido introducido como su primer carácter.

Programa 7.14

```
#include <iostream>
using namespace std;

int main()
{
  char primera_tecla, segunda_tecla;

  cout << "Mecanografie un caracter: ";
  cin.get(primera_tecla);
  cout << "La tecla que se acaba de aceptar es " << int(primera_tecla) << endl;
  cin.ignore();

  cout << "Mecanografie otro caracter: ";
  cin.get(segunda_tecla);
  cout << "La tecla que se acaba de aceptar es " << int(segunda_tecla) << endl;

  cin.ignore();
  return 0;
}
```

En el programa 7.14, hay que observar que cuando el usuario mecanografía la tecla m y oprime la tecla Entrar, la m es asignada a `primera_tecla` y el código para la tecla Entrar es ignorado. La siguiente llamada a `get()` almacena el código para la siguiente tecla oprimida en la variable `segunda_tecla`. Desde el punto de vista del usuario, la tecla Entrar no tiene efecto excepto para señalar el fin de cada entrada de carácter. La siguiente es una muestra de la ejecución del programa 7.14:

```
Mecanografie un caracter: m
La tecla que se acaba de aceptar es 109
Mecanografie otro caracter: b
La tecla que se acaba de aceptar es 98
```

Una segunda mirada a la validación de entradas del usuario

Como se mencionó durante la primera mirada a la validación de entradas del usuario (sección 3.4), los programas que responden de manera efectiva a entradas inesperadas del usuario se conocen de manera formal como programas robustos y de manera informal como programas "a prueba de balas". El código que valida las entradas del usuario y asegura que un programa no produce resultados imprevistos debido a entradas inesperadas es una señal de un programa robusto bien construido. Uno de sus trabajos, como programador, es producir tales programas. Para ver cómo pueden ocurrir resultados imprevistos, considérense los dos ejemplos de código que se presentan a continuación. Primero, supóngase que el programa contiene las siguientes instrucciones:

```
cout << "Introduzca un número entero: ";
cin >> valor;
```

Ahora supóngase que, por error, un usuario introduce los caracteres e4. En versiones anteriores de C++, esto causaría que el programa terminara en forma inesperada, o **se cayera**. Aunque todavía puede ocurrir una caída con el estándar ANSI/ISO actual (véase, por ejemplo, el ejercicio 9), no ocurrirá en este caso. Más bien, se asignará un valor entero sin sentido a la variable valor. Esto, por supuesto, invalidará los resultados obtenidos usando esta variable.

Como un segundo ejemplo, considérese el siguiente código, el cual causará que ocurra un ciclo infinito si el usuario introduce un valor no numérico (el programa puede detenerse oprimiendo el botón Ctrl y, mientras se mantiene oprimido, oprimiendo la tecla c):

```
double valor;

  do
  {
    cout << "Introduzca un número (introduzca 0 para salir): ";
    cin >> valor;

    cout << "La raiz cuadrada de este número es: " << sqrt(valor) << endl;
    }While (value !=0);
```

La técnica básica para manejar la introducción de datos inválidos y prevenir que código aparentemente inocuo, como en los dos ejemplos anteriores, produzca resultados imprevistos se conoce como ***validación de las entradas del usuario***. Esto significa validar los datos introducidos durante o después de la entrada de datos y proporcionar al usuario una forma de reintroducir los datos inválidos. La validación de las entradas del usuario es una parte esencial de cualquier programa viable desde el punto de vista comercial y, si se hace en forma correcta, protege al programa e inhibe el intento de procesar tipos de datos que pueden causar que un programa se caiga, cree ciclos infinitos o produzca otros resultados inválidos.

El elemento central en la validación de las entradas del usuario es la comprobación de cada carácter introducido para verificar que es un carácter legítimo para el tipo de datos esperado. Por ejemplo, si se requiere un número entero, los únicos caracteres aceptables son un signo de más (+) o de menos (−) a la izquierda y los dígitos 0 a 9. Estos caracteres pueden comprobarse conforme son mecanografiados, lo cual significa que la función `get()` se usa para introducir un carácter a la vez, o después que todos los caracteres son aceptados en una cadena, cada carácter de la cadena es comprobado en su validez. Una vez que han sido validados todos los caracteres introducidos, la cadena puede convertirse en el tipo de datos correcto.

Existen dos medios básicos para lograr la validez de los caracteres introducidos. Al principio de la sección 7.4 se presenta una de estas formas: la comprobación carácter por carácter. Una segunda técnica, la cual abarca un alcance más amplio de tareas de procesamiento de datos usando el manejo de excepciones, se presenta al final de la sección 7.4.

Ejercicios 7.3

1. Introduzca y ejecute el programa 7.10.
2. Introduzca y ejecute el programa 7.11.
3. Escriba un programa en C++ que cuente el número de palabras en una cadena. Una palabra se localiza siempre que se encuentra una transición de un espacio en blanco a un carácter que no está en blanco. Suponga que la cadena sólo contiene palabras separadas por espacios en blanco.
4. Genere diez números aleatorios en el rango de 0 a 129. Si el número representa un carácter imprimible, imprima el carácter con un mensaje apropiado que indique lo siguiente:

   El carácter es una letra minúscula
   El carácter es una letra mayúscula
   El carácter es un dígito
   El carácter es un espacio

   Si el carácter no es ninguno de estos datos, despliegue su valor en formato de número entero.

5. **a.** Escriba una función, `length()`, que determine y devuelva la longitud de una cadena sin usar el método de la clase de cadena `length()`.
   b. Escriba una función `main()` simple para probar la función `length()` escrita para el ejercicio 5a.
6. **a.** Escriba una función, `countlets()`, que devuelva el número de letras en una cadena transmitida como un argumento. Los dígitos, espacios, signos de puntuación, tabuladores y caracteres de línea nueva no deberán incluirse en la cuenta devuelta.
   b. Incluya el método `countlets()` escrito para el ejercicio 6a en un programa C++ ejecutable y use el programa para probar el método.
7. Escriba un programa que acepte una cadena de la consola y despliegue el equivalente hexadecimal de cada carácter en la cadena.
8. Escriba un programa en C++ que acepte una cadena de la consola y despliegue la cadena una palabra por línea.
9. En respuesta al siguiente código suponga que un usuario introduce los datos 12e4:

```
cout << "Introduzca un número entero: ";
cin >> valor;
```

   ¿Qué valor será almacenado en la variable entera `valor`?

10. a. Escriba un programa en C++ que deje de leer una línea de texto cuando se introduce un punto y despliegue el enunciado con un espaciado y uso de mayúsculas correctos. Para este programa, el espaciado correcto significa que sólo habrá un espacio entre palabras y que todas las letras deberán estar en minúsculas, excepto la primera letra. Por ejemplo, si el usuario introdujo el texto voy `a Ir AL ciNe`., el enunciado desplegado deberá ser `Voy a ir al cine.`

b. Determine qué caracteres, si es que hay alguno, no fueron desplegados en forma correcta por el programa que creó para el ejercicio 10a.

11. Escriba un programa en C++ que acepte un nombre como nombre y apellido y luego lo despliegue como apellido, nombre. Por ejemplo, si el usuario introdujo Gary Bronson, la salida deberá ser Bronson, Gary.

12. Modifique el programa escrito para el ejercicio 11 para incluir una serie de cinco nombres.

7.4 VALIDACIÓN DE DATOS DE ENTRADA

Uno de los usos principales de las cadenas en los programas es la validación de las entradas del usuario. La necesidad de validar las entradas del usuario es esencial: aunque un programa le indique al usuario que introduzca un tipo específico de datos, como un número entero, esto no asegura que el usuario lo hará. Lo que un usuario introduce, de hecho, está totalmente fuera del control del programador. Lo que sí se encuentra bajo control es cómo tratar los datos introducidos.

Por supuesto que no es bueno decirle a un usuario frustrado "El programa claramente le indicó que introdujera un número entero y usted introdujo una fecha". En cambio, los programas exitosos siempre anticipan los datos inválidos y los aíslan para que no sean aceptados y procesados. Esto se logra por lo general validando primero que los datos sean del tipo correcto; si así es, los datos son aceptados; de lo contrario, se le solicita al usuario que los reintroduzca, proporcionando una explicación de por qué son inválidos los datos introducidos.

Uno de los métodos más comunes para validar la introducción de datos numéricos es aceptar todos los números como cadenas. Cada carácter en la cadena puede ser comprobado entonces para asegurar que cumple con el tipo de datos solicitado. Después que se hace esta comprobación y los datos son verificados de acuerdo con el tipo correcto, la cadena se convierte en un valor de un número entero o de precisión doble usando las funciones de conversión enlistadas en la tabla 7.7. (Para datos aceptados usando objetos de la clase `string`, debe aplicarse el método `c_str()` a la cadena antes de invocar a la función de conversión.)

Como ejemplo, considérese la introducción de un número entero. Para ser válidos, los datos introducidos deben apegarse a las siguientes condiciones:

- Los datos deben contener al menos un carácter.
- Si el primer carácter es un signo + o −, los datos deben contener al menos un dígito.
- Sólo dígitos de 0 a 9 después del primer carácter son aceptables.

Tabla 7.7 Funciones de conversión c-string

Función	Descripción	Ejemplo
`int atoi(stringExp)`	Convierte `stringExp` en un número entero. La conversión se detiene en el primer carácter que no es un número entero.	`atoi("1234")`
`double atof(stringExp)`	Convierte `stringExp` en un número de precisión doble. La conversión se detiene en el primer carácter que no puede interpretarse como un doble.	`atof("12.34")`
`char[] itoa(integerExp)`	Convierte `integerExp` en un arreglo de caracteres. El espacio asignado para los caracteres devueltos debe ser lo bastante grande para el valor convertido.	`itoa(1234)`

La siguiente función, `isvalidInt(),` puede usarse para verificar que una cadena introducida cumple con estas condiciones. Esta función devuelve el valor booleano `verdadero` si las condiciones se satisfacen; de lo contrario, devuelve un valor booleano `falso.`

```
bool isvalidInt(string str)
{
  int inicio = 0;
  int i;
  bool valido = verdadero; // asume un válido
  bool signo = falso; // asume que no es signo


  // comprueba una cadena vacia
  if (int(str.length()) == 0) válido = falso;

  // comprueba un signo a la izquierda
  if (str.at(0) == '-'|| str.at(0) == '+')
  {
    signo = verdadero;
    inicio = 1; // comienza a comprobar los dígitos después del signo
  }

  // comprueba que hay al menos un carácter después del signo
  if (signo && int(str.length() == 1)) valido = falso

  // ahora comprueba la cadena, la cual sabemos que tiene al menos un ca-
     rácter que no es un signo
  i = inicio;
```

```
  while(valido && i < int(str.length()))
  {
    if(!isdigit(str.at(i))) valido = falso; // encuentra un carácter que
    no es digito
    i++; // se mueve al siguiente carácter
  }

  return valido;
}
```

En el código para el método `isvalidInt()`, debe ponerse atención a las condiciones que se están comprobando. Éstas se comentan en el código y consisten en comprobar lo siguiente:

- La cadena no está vacía.
- Un símbolo de signo válido (+ o −) está presente.
- Si un símbolo de signo está presente, al menos un dígito lo sigue.
- Todos los caracteres restantes en la cadena son dígitos.

Sólo si todas estas condiciones se cumplen la función devuelve un valor booleano `verdadero`. Una vez que es devuelto este valor, la cadena puede ser convertida con seguridad en un número entero con la tranquilidad que no resultará ningún valor inesperado que obstaculice el procesamiento de datos posterior. El programa 7.15 usa este método dentro del contexto de un programa completo.

Programa 7.15

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  bool isvalidInt(string); // prototipo de la función (declaración)
  string valor;
  int numero;

  cout << "Introduzca un número entero: ";
  getline(cin, valor);
```

(Continúa)

(Continuación)

```
  if (!isvalidInt(valor))
   cout << "El número que introdujo no es un número entero válido.";
  else
  {
      numero = atoi(valor.c_str());
    cout << "El número entero que introdujo es " << número;
  }


  return 0;
}

bool isvalidInt(string str)
{
  int inicio = 0;
  int i;
  bool valido = verdadero; // asume un número valido
  bool signo = falso; // asume que no es signo


  // comprueba una cadena vacía
  if (int(str.length()) == 0) valido = falso;

  // comprueba un signo a la izquierda
  if (str.at(0) == '-'|| str.at(0) == '+')
  {
    signo = verdadero;
    inicio = 1; // comienza a comprobar los dígitos después del signo
  }

  // comprueba que hay al menos un carácter después del signo
  if (signo && int(str.length() == 1)) valido = falso

  // ahora comprueba la cadena, la cual sabemos que tiene al menos un
     carácter que no es un signo
  i = inicio;
  while(valido && i < int(str.length()))
  {
    if(!isdigit(str.at(i))) valido = falso; // encuentra un carácter
    que no es dígito

    i++; // se mueve al siguiente carácter
  }

  return valido;
}
```

Dos muestras de ejecuciones del programa 7.15 produjeron lo siguiente:

```
Introduzca un número entero: 12e45
El numero que introdujo no es un número entero válido.
```

y

```
Introduzca un número entero: -12345
El número entero que introdujo es -12345
```

Como lo ilustra esta salida, el programa determina con éxito que se introdujo un carácter inválido en la primera ejecución.

Una segunda línea de defensa es proporcionar código de procesamiento de errores dentro del contexto del código de manejo de excepciones. Este tipo de código se proporciona por lo general para permitir al usuario corregir un problema como la introducción de datos inválidos al reintroducir un nuevo valor. El medio para proporcionar esto en C++ se conoce como manejo de excepciones y se presenta a continuación.

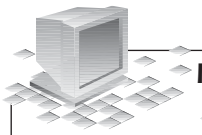
Usando el manejo de excepciones, puede construirse un medio completo para asegurar que el usuario introduce un número entero en respuesta a una solicitud de un valor entero. La técnica que se usará extiende la función `isvalidInt()` incluida en el programa 7.15 para asegurar que un valor entero inválido no sólo es detectado, sino que el programa proporciona al usuario la opción de reintroducir valores hasta que se obtiene un número entero válido. Esta técnica puede aplicarse con facilidad para asegurar la introducción de un número de precisión doble válido, el cual es el otro tipo de datos numéricos solicitado con frecuencia como datos introducidos por el usuario.

Empleando la función `isvalidInt()` proporcionada en el programa 7.15, ahora desarrollamos una función más detallada llamada `getanInt()` que usa el procesamiento de excepciones para aceptar en forma continua una entrada del usuario hasta que se detecta una cadena que corresponde a un número entero válido. Una vez que se ha introducido dicha cadena, la función `getanInt()` convierte la cadena en un número entero y devuelve el valor entero. Esto asegura que el programa que solicita un número entero en realidad recibe un número entero y previene cualesquiera efectos indebidos, como una caída del programa debida a que se introdujo un tipo de datos inválido.

El algoritmo que se usará para llevar a cabo esta tarea es:

```
Establecer una variable booleana llamada notanint como verdadero
while (notanint es verdadero)
    try
        Aceptar un valor de cadena
        Si el valor de cadena no corresponde a un número entero se
        lanza una excepción
    atrapar la excepción
        Desplegar el mensaje de error "Número entero inválido - Por fa-
        vor reintroduzca: "
        Enviar el control de vuelta a la instrucción while
        Establecer notanint en falso (esto causa que termine el ciclo)
  End while
  Devolver el número entero correspondiente a la cadena introducida
```

El código correspondiente a este algoritmo se resalta en el programa 7.16.



Programa 7.16

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  int getanInt();  // declaración de función (prototipo)
  int valor;

  cout << "Introduzca un valor entero: ";
  valor = getanInt();
  cout << "El número entero introducido es: " << valor << endl;

  return 0;
}
```

```
int getanInt()
{
  bool isvalidInt(string); // declaración de función (prototipo)
  bool notanint = verdadero;
  string segundo_valor;

  while (notanint)
  {
    try
    {
      cin >> segundo_valor; // acepta una entrada de cadena
      if (!isvalidInt(segundo_valor)) throw segundo_valor;
      }
    catch (string e)
    {
      cout << "Número entero inválido - Por favor reintroduzca: ";
        continue; // envia el control a la instruccion while
    }
    notanint = falso;
  }
  return atoi(segundo_valor.c_str()); // convierte en un número entero
}
```

(Continúa)

(Continuación)

```
bool isvalidInt(string str)
{
  int inicio = 0;
  int i;
  bool valido = verdadero;  // supone un número válido
  bool signo = falso;  // supone que no es signo

  // comprueba una cadena vacía
  if (int(str.length()) == 0) válido = falso;

  // comprueba un signo a la izquierda
  if (str.at(0) == '-'|| str.at(0) == '+')
  {
    signo = verdadero;
    inicio = 1;  // inicia la comprobación de digitos después del signo
  }

  // comprueba que haya al menos un carácter después del signo
  if (signo && int(str.length()) == 1) válido = falso;

  // ahora comprueba la cadena, la cual sabemos que al menos tiene un
     carácter que no es un signo
  i = inicio;
  while(valido && i < int(str.length()))
  {
    if(!isdigit(str.at(i))) valido = falso;  // encuentra un carácter
    que no es un dígito
    i++;  // se mueve al siguiente carácter
  }

 return valido;
}
```

A continuación hay una muestra de la salida producida por el programa 7.16:

```
Introduzca un valor entero: abc
Número entero inválido – Por favor reintroduzca: 12.
Número entero inválido – Por favor reintroduzca: 12e
Número entero inválido – Por favor reintroduzca: 120
El número entero introducido es: 120
```

Como lo muestra esta salida, la función `getanInt()` funciona en forma correcta. Solicita entrada en forma continua hasta que se introduce un número entero válido.

Ejercicios 7.4

1. Escriba un programa en C++ que indique al usuario que escriba un número entero. Haga que su programa acepte el número, como un entero, usando `cin` y, usando `cout,` despliegue el valor que aceptó en realidad su programa de los datos introducidos. Ejecute su programa cuatro veces. La primera vez que lo haga, introduzca un número entero válido, la segunda vez introduzca un número de precisión doble y la tercera vez introduzca un carácter. A continuación, introduzca el valor 12e34.
2. Modifique el programa que escribió para el ejercicio 1 pero haga que su programa use una variable de precisión doble. Ejecute el programa cuatro veces. La primera vez introduzca un número entero, la segunda vez introduzca un número decimal, la tercera vez introduzca un número decimal con una f como el último carácter introducido y la cuarta vez introduzca un carácter. Usando el despliegue de la salida, haga el seguimiento de cuál número aceptó en realidad su programa de los datos que introdujo. ¿Qué sucedió, si es que sucedió algo, y por qué?
3. **a.** ¿Por qué piensa que los programas de aplicación exitosos contienen comprobaciones extensas de la validez de los datos introducidos? (*Sugerencia:* Revise los ejercicios 1 y 2.)
   b. ¿Cuál piensa que sea la diferencia entre la comprobación de un tipo de datos y una comprobación de la racionalidad de los datos?
   c. Suponga que un programa solicita que el usuario introduzca un día, mes y año. ¿Cuáles son algunas comprobaciones razonables que podrían hacerse con los datos introducidos?
4. **a.** Introduzca y ejecute el programa 7.15.
   b. Ejecute el programa 7.15 cuatro veces, usando los datos a los que se hizo referencia en el ejercicio 1 para cada ejecución.
5. Modifique el programa 7.15 para desplegar los caracteres inválidos que se introduzcan.
6. Modifique el programa 7.15 para solicitar un número entero de manera continua hasta que se introduzca un número válido.
7. Modifique el programa 7.15 para eliminar todos los espacios antes y después de la cadena introducida antes que se compruebe su validez.
8. Escriba una función que compruebe cada dígito conforme se introduzca, en lugar de comprobar la cadena completada, como se hace en el programa 7.15.
9. Introduzca y ejecute el programa 7.16.
10. Modifique la función `isvalidInt()` usada en el programa 7.16 para eliminar todos los espacios en blanco antes y después de su argumento de cadena antes de determinar si la cadena corresponde a un número entero válido.
11. Modifique la función `isvalidInt()` usada en el programa 7.16 para aceptar una cadena que termine en un punto decimal. Por ejemplo, introduzca 12. Deberá aceptarse y convertirse en el número entero doce.
12. **a.** Escriba una función de C++ llamada `isvalidReal()` que compruebe un número de precisión doble válido. Esta clase de número puede tener un signo + o – opcional, cuando más un punto decimal, el cual puede ser el primer carácter,

y cuando menos un dígito entre 0 y 9 inclusive. La función deberá devolver un valor booleano de verdadero si el número introducido es un número real; de lo contrario, deberá devolver un valor booleano de falso.

b. Modifique la función `isvalidReal()` escrita para el ejercicio 12a para eliminar todos los espacios en blanco antes y después de su argumento de cadena antes de determinar si la cadena corresponde a un número real válido.

13. Escriba y ejecute una función de C++, llamada `getareal()`, que use el manejo de excepciones para aceptar en forma continua la introducción de una cadena hasta que se introduzca una cadena y pueda convertirse en un número real. La función deberá devolver un valor doble correspondiente al valor de cadena introducido por el usuario.

7.5 Espacio de nombres y creación de una biblioteca personal

Hasta la introducción de las computadoras personales a principios de la década de los años 80, con el uso extenso de circuitos integrados y microprocesadores, tanto la velocidad de las computadoras como su memoria disponible estaban muy restringidas. Por ejemplo, las computadoras más avanzadas de la época tenían velocidades medidas en milisegundos (una milésima de segundo); las computadoras actuales tienen velocidades que se miden en nanosegundos (una mil millonésima de segundo) y superiores. Del mismo modo, la capacidad de memoria de las primeras computadoras de escritorio consistía en 32 mil ubicaciones, con cada ubicación consistente en ocho bits. Las memorias de las computadoras actuales consisten en millones de ubicaciones de memoria, cada una consistente de 32 a 64 bits.

Estas primeras restricciones de hardware hicieron imperativo que los programadores utilizaran todos los trucos posibles para ahorrar espacio de memoria y hacer que los programas se ejecutaran de manera más eficiente. Casi todos los programas se hacían a mano e incluían lo que se conocía como "código inteligente" para minimizar el tiempo de ejecución y maximizar el uso del almacenamiento de memoria. Por desgracia, este código individualizado, con el tiempo, se convirtió en un desastre. Los nuevos programadores tuvieron que dedicar un tiempo considerable para entender el código existente; incluso el programador original tenía problemas para comprender código que había escrito sólo unos meses antes. Esto hacía que las modificaciones consumieran mucho tiempo y fueran muy costosas, e impidió el uso rentable del código existente para nuevas instalaciones.

La incapacidad para reutilizar el código en forma eficiente, combinada con la expansión de las capacidades del hardware, proporcionó el incentivo para instaurar una programación más eficiente. Al principio esto condujo a los conceptos de programación estructurada incorporados en los lenguajes por procedimientos, como Pascal, y en la actualidad a las técnicas orientadas a objetos que forman la base de C++. Sin embargo, una de las primeras críticas a C++ fue que no proporcionaba una biblioteca de clases detallada. Esto ha cambiado en forma impresionante con el estándar ANSI/ISO actual y la inclusión de una biblioteca C++ muy amplia.

No obstante, sin importar cuántas clases y métodos proporcionen, cada tipo de aplicación de programación, como en las áreas de ingeniería, científica y financiera, tienen sus propios requerimientos especializados. Por ejemplo, C++ proporciona funciones de fecha y hora bastante aceptables en su archivo de encabezado `ctime`. Sin embargo, para necesidades especiales, como las que se encuentran en problemas de organización, estas fun-

ciones deben expandirse. Por tanto, un conjunto de funciones más completo incluiría la capacidad de encontrar el número de días hábiles entre dos fechas y que tomara en cuenta fines de semana y días festivos. También requeriría funciones que aplicaran algoritmos del día anterior y el siguiente y tomaran en cuenta los años bisiestos y los días reales en cada mes. Éstos podrían proporcionarse como parte de una clase `Date` más completa o como funciones que no son de clase.

En situaciones como ésta, los programadores crean y comparten sus propias bibliotecas de clases y funciones con otros programadores que trabajan en los mismos proyectos u otros similares. Una vez que se han probado las clases y funciones, pueden incorporarse en cualquier programa sin dedicar más tiempo a la codificación.

En esta etapa en su carrera de programación, puede comenzar a construir su propia biblioteca de funciones y clases especializadas. La sección 7.4 describe cómo puede lograrse esto usando las funciones de validación de entradas, `isvalidInt()` y `getanInt()`, las cuales se reproducen a continuación por comodidad:

```
bool isvalidInt(string str)
{
  int inicio = 0;
  int i;
  bool valido = verdadero;  // supone un número valido
  bool signo = falso;  // supone que no es signo

  // comprueba una cadena vacía
  if (int(str.length()) == 0) valido = falso;

  // comprueba un signo a la izquierda
  if (str.at(0) == '-'|| str.at(0) == '+')
  {
    signo = verdadero;
    inicio = 1; // inicia la comprobación de dígitos después del signo
  }

  // comprueba que haya al menos un carácter después del signo
  if (signo && int(str.length()) == 1) valido = falso;

  // ahora comprueba la cadena, la cual sabemos que al menos tiene un ca-
     rácter que no es un signo
  i = inicio;
  while(valido && i < int(str.length()))
  {
    if(!isdigit(str.at(i))) valido = falso; // encuentra un carácter que
    no es un dígito
    i++; // se mueve al siguiente carácter
  }

  return valido;
}
int getanInt()
{
  bool isvalidInt(string); // declaración de función (prototipo)
  bool notanint = verdadero;
  string segundo_valor;
```

```
  while (notanint)
  {
    try
    {
      cin >> segundo_valor; // acepta una entrada de cadena
      if (!isvalidInt(segundo_valor)) throw segundo_valor;
    }
    catch (string e)
    {
      cout << "Número entero inválido - Por favor reintroduzca: ";
         continue; // envia el control a la instruccion while
    }
    notanint = falso;
  }
  return atoi(segundo_valor.c_str()); // convierte en un número entero
}
```

El primer paso en la creación de una biblioteca es encapsular todas las funciones y clases preferidas en uno o más espacios de nombres (namespaces) y luego almacenar el código completo (usando o no un espacio de nombres) en uno o más archivos. Por ejemplo, puede crear un espacio de nombres, `comprDatos`, y guardarlo en el archivo llamado `comprDatos.cpp`. Es importante señalar que el nombre de archivo bajo el cual se guarda el espacio de nombres no necesita ser el mismo que el nombre del espacio de nombres usado en el código.

La sintaxis para crear un espacio de nombres es la siguiente:

```
namespace name
{
  aquí van las funciones y/o clases
}  // end of namespace
```

Incluir las dos funciones `isvalidInt()` y `getanInt()` dentro de un espacio de nombres, `comprDatos`, y agregar los archivos `include` apropiados y la instrucción de declaración `using` necesaria para el espacio de nombres nuevo produjo el siguiente código. Por comodidad, la sintaxis requerida para crear el espacio de nombres se ha resaltado:

```
#include <iostream>
#include <string>
using namespace std;

namespace comprDatos
{
  bool isvalidInt(string str)
  {
    int inicio = 0;
    int i;
    bool valido = verdadero;  // supone un número válido
    bool signo = falso;       // supone que no es signo
```

```
    // comprueba una cadena vacía
    if (int(str.length()) == 0) valido = falso;

    // comprueba un signo a la izquierda
    if (str.at(0) == '-'|| str.at(0) == '+')
    {
      signo = verdadero;
      inicio = 1; // inicia la comprobación de dígitos después del signo
    }

    // comprueba que haya al menos un carácter después del signo
    if (signo && int(str.length()) == 1) valido = falso;

    // ahora comprueba la cadena, la cual sabemos que al menos tiene un ca-
       rácter que no es un signo
    i = inicio;
    while(valido && i < int(str.length()))
    {
      if(!isdigit(str.at(i))) valido = falso;  // encuentra un caracter que
      no es un dígito
      i++;  // se mueve al siguiente carácter
    }
    return valido;
  }

  int getanInt()
  {
    bool isvalidInt(string); // declaración de función (prototipo)
    bbool notanint = verdadero;
    string segundo_valor;

    while (notanint)
    {
      try
      {
        cin >> segundo_valor; // acepta una entrada de cadena
        if (!isvalidInt(segundo_valor)) throw segundo_valor;
      }
      catch (string e)
      {
        cout << "Número entero invalido - Por favor reintroduzca: ";
        continue; // envía el control a la instrucción while
      }
      notanint = falso;
    }
    return atoi(segundo_valor.c_str()); // convierte en un número entero
  }
} // fin del namespace comprDatos
```

Una vez que se ha creado el espacio de nombres y se ha almacenado en un archivo, puede incluirse dentro de otro archivo al suministrar una directiva preprocesadora para informar al compilador dónde se encuentra el espacio de nombres deseado y al incluir una directiva `using` que instruya al compilador sobre cuál espacio de nombres particular en el archivo usar. Para el espacio de nombres `comprDatos`, el cual está almacenado en un archivo llamado `comprDatos.cpp`, esto se logra mediante las siguientes instrucciones:

```
#include <c:\\mibiblioteca\\comprDatos>
using namespace comprDatos;
```

La primera instrucción proporciona el nombre de la ruta completa para el archivo del código fuente. Hay que observar que se ha usado un nombre de ruta completo y que se usan dos diagonales inversas para separar los nombres de la ruta. Las diagonales dobles inversas se requieren siempre que se proporciona un nombre de ruta relativo o completo. La única ocasión en que no se requieren las diagonales invertidas es cuando el código de la biblioteca reside en el mismo directorio que el programa que se está ejecutando. Como se indicó, el archivo fuente `comprDatos` está guardado dentro de una carpeta llamada `mibiblioteca`. La segunda instrucción le indica al compilador que use el espacio de nombres `comprDatos` dentro del archivo designado. El programa 7.17 incluye estas dos instrucciones dentro de un programa ejecutable.

Programa 7.17

```
#include <c:\\mibiblioteca\\comprDatos.cpp>
using namespace comprDatos;

int valor;
{
  int value;

  cout << "Introduzca un valor entero: ";
  valor = getanInt();
  cout << "El número entero introducido es: " << valor << endl;



  return 0;
}
```

El único requisito para la instrucción `include` en el programa 7.17 es que el nombre y la ubicación del archivo deben corresponder a un archivo existente que tenga el mismo nombre en la ruta designada; de lo contrario, ocurrirá un error de compilador. Si desea nombrar el archivo del código fuente utilizando una extensión de archivo, puede usarse cualquier extensión en tanto se mantengan las siguientes reglas:

1. El nombre del archivo bajo el cual está almacenado el código incluye la extensión.
2. El mismo nombre de archivo, incluyendo la extensión, se usa en la instrucción `include`.

Por tanto, si el nombre de archivo usado para almacenar las funciones fuera `biblDatos.cpp`, la instrucción `include` en el programa 7.17 sería

```
#include <c:\\mibiblioteca\\biblDatos.cpp>
```

Además, no se requiere un espacio de nombres dentro del archivo. Usar un espacio de nombres nos permite aislar las funciones de comprobación de datos en un área y agregar espacios de nombres adicionales al archivo cuando sea necesario. La designación de un espacio de nombres en la instrucción `using` le indica al compilador que incluya sólo el código en el espacio de nombres especificado, en lugar de todo el código en el archivo. En el programa 7.17, si las funciones de comprobación de datos no estuvieran encerradas dentro de un espacio de nombres, se habría omitido la instrucción `using` para el espacio de nombres `comprDatos`.

Incluir las funciones de comprobación de datos escritas y probadas con anterioridad dentro del programa 7.17 como un archivo separado le permite enfocarse en el código dentro del programa que usa estas funciones, en lugar de estar preocupado con el código de la función en sí. Esto le permite concentrarse en el uso de estas funciones en vez de reexaminar o verificar el código de función escrito y probado antes. En el programa 7.17, el método `main()` ejerce las funciones de comprobación de datos y produce la misma salida que el programa 7.16. Al crear el espacio de nombres `comprDatos`, se ha incluido código fuente para las dos funciones. Esto no se requiere, y en cambio puede guardarse una versión compilada del código fuente. Por último, a un espacio de nombres definido en un archivo pueden hacérseles adiciones en otro archivo usando el mismo nombre del espacio de nombres en el archivo nuevo e incluyendo una instrucción `using` para el espacio de nombres del primer archivo.

Ejercicios 7.5

1. Introduzca y compile el programa 7.17. (*Sugerencia:* El archivo de encabezado de espacio de nombres `comprDatos` y el archivo del programa están disponibles con el código fuente proporcionado en el sitio web de Course Technology para este texto. Véase el ejercicio 4 para el procedimiento de descarga.)

2. ¿Por qué un programador suministraría un archivo de espacio de nombres en su forma compilada y no como código fuente?

3. **a.** ¿Cuál es la ventaja de los espacios de nombres?
   b. ¿Cuál es una posible desventaja de los espacios de nombres?

4. ¿Qué tipos de clases y funciones incluiría en una biblioteca personal? ¿Por qué?

5. **a.** Escriba una función de C++, `entero()`, que devuelve la parte entera de cualquier número transmitido a la función. (*Sugerencia:* Asigne el argumento transmitido a una variable de número entero.)
   b. Incluya la función escrita en el ejercicio 5a en un programa que funcione. Asegúrese que su función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.
   c. Cuando esté seguro que la función `entero()` escrita para el ejercicio 5a funciona en forma correcta, guárdela en un espacio de nombres y en una biblioteca personal de su elección.

6. **a.** Escriba una función de C++, `partefracc()`, que devuelva la parte fraccionaria de cualquier número transmitido a la función. Por ejemplo, si se transmite el número 256.879 a `partefracc()`, deberá devolverse el número .879. Haga que la función `partefracc()` llame a la función `entero()` que escribió en el ejercicio 5a. El número devuelto puede determinarse entonces como el número transmitido a `partefracc()` menos el valor devuelto cuando el mismo argumento es transmitido a `entero()`.
   b. Incluya la función escrita en el ejercicio 6a en un programa que funcione. Asegúrese que la función es llamada desde `main()` y devuelve en forma correcta un valor a `main()`. Haga que `main()` use una instrucción `cout` para desplegar el valor devuelto. Pruebe la función transmitiéndole varios datos.
   c. Cuando esté seguro que la función `partefracc()` escrita para el ejercicio 6a funciona en forma correcta, guárdela en el mismo espacio de nombres y biblioteca personal seleccionados para el ejercicio 5c.

7.6 Errores comunes de programación

Aquí se muestran los errores más comunes asociados con la definición y procesamiento de cadenas:

1. Se ha olvidado incluir el archivo de encabezado `string` cuando se usan objetos de la clase `string`.
2. Se ha olvidado que el carácter de línea nueva, `'\n'`, es un carácter de introducción de datos válido.
3. Se ha olvidado convertir un objeto de clase `string` usando el método `c_str()` cuando se convierten objetos de la clase `string` en tipos de datos numéricos.

7.7 Resumen del capítulo

1. Una cadena literal es cualquier secuencia de caracteres encerrados entre comillas. Una cadena literal se conoce como un valor de cadena, una constante de cadena y, de manera más convencional, una cadena.
2. Una cadena puede construirse como un objeto de la clase `string`.
3. La clase `string` se usa por lo común para construir cadenas con propósitos de entrada y salida, como para indicadores y mensajes desplegados. Debido a sus capacidades, esta clase se usa cuando es necesario comparar o buscar las cadenas o los caracteres individuales en una cadena y necesitan examinarse o extraerse como una subcadena. En situaciones más avanzadas se usa cuando es necesario reemplazar, insertar o eliminar en forma regular los caracteres dentro de una cadena.
4. Las cadenas pueden manipularse usando los métodos de la clase de la que son objetos o usando la cadena de propósito general y métodos de carácter.
5. El objeto `cin`, por sí mismo, tiende a ser de utilidad limitada para la introducción de cadenas debido a que termina la entrada cuando encuentra un espacio en blanco.
6. Para introducir datos a la clase `string` se usa el método `getline()`.
7. El objeto `cout` puede usarse para desplegar las cadenas de la clase `string`.

Consideración de opciones de carrera

Ciencia de materiales e ingeniería metalúrgica

En gran medida, los avances en muchas áreas de la ingeniería en el siglo XX fueron posibles por el descubrimiento de nuevos materiales y una mejor comprensión de las propiedades de los materiales existentes. El conocimiento de los principios físicos y químicos que determinan las propiedades eléctricas de materiales exóticos llamados semiconductores ha producido un progreso fantástico en el campo de los dispositivos de estado sólido, desde transistores hasta chips de circuitos integrados para mainframes. La mejor comprensión de los orígenes de las propiedades metálicas como dureza, fuerza, ductilidad, corrosividad y otras han conducido a una mejora en el diseño de automóviles, aviones, naves espaciales y todo tipo de maquinarias. El campo se subdivide básicamente en metales y no metales, aunque con frecuencia existe una considerable superposición de intereses y actividades.

Ciencia de materiales

La ciencia de materiales se interesa en el comportamiento y propiedades de los materiales, tanto metales como no metales, desde las perspectivas microscópica y macroscópica. Incluye las siguientes áreas:

1. Cerámica. Materiales que no son cristalinos, como el vidrio, que son no metálicos y que requieren altas temperaturas en su procesamiento. La cerámica puede hacerse quebradiza o flexible, dura o suave, o más fuerte que el acero. Puede lograrse que tenga una gran variedad de propiedades químicas.
2. Polímeros. Propiedades estructurales y físicas de polímeros orgánicos, inorgánicos y naturales que son útiles en aplicaciones de ingeniería.
3. Fabricación, procesamiento y tratamiento de materiales. Todos los aspectos de la manufactura de cerámica, metales y síntesis de polímeros, desde el crecimiento de cristales y fibras hasta la formación de metales.
4. Corrosión. Mecanismo de reacción y termodinámica de la corrosión de metales en la atmósfera o sumergidos bajo agua o sustancias químicas, sean fijos o bajo tensión.
5. Tensión-estiramiento y fatiga-fractura de materiales de ingeniería. Las propiedades físicas que rigen la deformación y fractura de materiales y su mejoramiento y uso en la construcción y el diseño.

(continúa)

Consideración de opciones de carrera

Ingeniería metalúrgica

La ingeniería metalúrgica es la rama de la ingeniería responsable de la producción de metales y aleaciones metálicas, desde el descubrimiento de depósitos de menas hasta la fabricación del metal refinado en productos útiles. Los ingenieros metalúrgicos son importantes en todos los pasos de la producción de metales a partir de las menas de metal. La ingeniería metalúrgica incluye las siguientes áreas:

1. Ingeniería de minas. Por lo general una rama separada de la ingeniería. Sin embargo, los intereses de los ingenieros de minas y los metalúrgicos con frecuencia se superponen en los procesos de extracción de metales de las menas de metal y el refinamiento para hacer productos utilizables. La metalurgia de extracción usa las reacciones físicas y químicas para optimizar la producción de metal.
2. Fabricación de metales. Formación de metales en productos como latas, cables y tubos; moldeado y unión de metales; por ejemplo, por soldadura.
3. Metalurgia física. Análisis de las características de tensión-estiramiento, fatiga-fractura de los metales y aleaciones metálicas para prevenir fallas en los componentes de ingeniería.

Parte dos

Programación orientada a objetos

Capítulos

8 Flujos de archivos de E/S y archivos de datos

9 Introducción a las clases

10 Funciones de clases y conversiones

CAPÍTULO 8

Flujos de archivos de E/S y archivos de datos

TEMAS

8.1 OBJETOS Y MÉTODOS EN EL FLUJO DE ARCHIVOS DE E/S
ARCHIVOS
OBJETOS DE FLUJO DE ARCHIVOS
MÉTODOS DE FLUJO DE ARCHIVOS
CERRAR UN ARCHIVO

8.2 LECTURA Y ESCRITURA DE ARCHIVOS BASADOS EN CARACTERES
LECTURA DE UN ARCHIVO DE TEXTO
ARCHIVOS ESTÁNDAR EN DISPOSITIVOS
OTROS DISPOSITIVOS

8.3 EXCEPCIONES Y COMPROBACIÓN DE ARCHIVOS
APERTURA MÚLTIPLE DE ARCHIVOS

8.4 ARCHIVOS DE ACCESO ALEATORIO

8.5 FLUJOS DE ARCHIVO COMO ARGUMENTOS DE FUNCIONES

8.6 ERRORES COMUNES DE PROGRAMACIÓN

8.7 RESUMEN DEL CAPÍTULO

8.8 COMPLEMENTO DEL CAPÍTULO: LA BIBLIOTECA DE CLASE `iostream`
MECANISMO DE TRANSFERENCIA DE FLUJO DE ARCHIVOS
COMPONENTES DE LA BIBLIOTECA DE CLASE `iostream`
FORMATEO EN MEMORIA

Los datos para los programas que se han utilizado hasta ahora se han asignado en forma interna dentro de los programas o han sido introducidos por el usuario durante la ejecución del programa. Como tales, los datos utilizados en estos programas son almacenados en la memoria principal de la computadora y dejan de existir una vez que el programa que los usa termina de ejecutarse. Este tipo de entrada de datos está bien para cantidades pequeñas de datos. Sin embargo, imagine a una compañía que tiene que pagarle a alguien para mecanografiar los nombres y direcciones de cientos o miles de clientes cada mes cuando se preparan y envían las facturas.

Como aprenderá en este capítulo, almacenar estos datos fuera de un programa en un medio de almacenamiento conveniente es más sensato. Los datos almacenados juntos bajo un nombre común en un medio de almacenamiento distinto a la memoria principal de la computadora se llaman archivos de datos. De manera

típica, los archivos de datos se almacenan en discos, cintas o CD-ROM. Además de proporcionar un almacenamiento permanente para los datos, los archivos de datos pueden ser compartidos entre programas, de modo que la salida de datos de un programa puede ser introducidos de manera directa en otro programa. Se comenzará este capítulo aprendiendo cómo se crean y mantienen los archivos de datos en C++. Una tarea muy importante sobre el uso de archivos de datos es asegurar que su programa abra y se conecte en forma correcta con ellos antes que comience cualquier procesamiento de datos. Por esta razón, aprenderá cómo usar el manejo de excepciones para esta tarea. Este tipo de detección y corrección de errores es de interés primordial en todos los programas escritos de manera profesional.

8.1 OBJETOS Y MÉTODOS EN EL FLUJO DE ARCHIVOS DE E/S

Para almacenar y recuperar datos fuera de un programa en C++, se necesitan dos cosas:

- Un archivo
- Un objeto de flujo de archivos

Aprenderá sobre estos temas importantes en las dos secciones siguientes.

Archivos

Un **archivo** es una colección de datos almacenados juntos bajo un nombre común, por lo general en un disco, cinta magnética o CD-ROM. Por ejemplo, los programas en C++ que se almacenan en un disco son ejemplos de archivos. Los datos almacenados en un archivo de programa son el código del programa que se convierte en datos de entrada para el compilador de C++. Sin embargo, En el contexto del procesamiento de datos, el programa en C++ por lo general no es considerado como datos, y el término "archivo", o "archivo de datos", por lo general sólo se refiere a archivos externos que contienen los datos usados en un programa en C++.

Un archivo se almacena en forma física en un medio externo como un disco. Cada archivo tiene un nombre de archivo único conocido como el **nombre externo** del archivo. El nombre externo es la manera en que es conocido el archivo por el sistema operativo. Cuando se revisa el contenido de un directorio o carpeta (por ejemplo, en el Explorador de Windows), se pueden observar los archivos enlistados por sus nombres externos. Cada sistema operativo de computadora tiene su propia especificación en cuanto al número máximo de caracteres permitidos para un nombre externo de archivo. La tabla 8.1 muestra estas especificaciones para los sistemas operativos más comunes.

Tabla 8.1 Cantidad máxima de caracteres permitidos en el nombre de un archivo

Sistema operativo	Largo máximo del nombre de archivo
DOS	8 caracteres más un punto opcional y una extensión de tres caracteres
Windows 98, 2000, XP	255 caracteres
UNIX Primeras versiones Versiones actuales	 14 caracteres 255 caracteres

Un poco de antecedentes

Privacidad, seguridad y archivos

Los archivos de datos existían mucho antes que se usaran las computadoras, pero se almacenaban sobre todo como registros en papel en los archiveros. Los términos como *abrir, cerrar, registros* y *consulta* que se usan en el manejo de archivos de computadora son remembranzas de las técnicas antiguas para tener acceso a archivos en papel almacenados en cajones.

En la actualidad, la mayor parte de los archivos se almacenan en forma electrónica, y la cantidad de información recopilada y almacenada prolifera. La facilidad para compartir grandes cantidades de datos en forma electrónica ha conducido al aumento en los problemas de privacidad y seguridad.

Cada vez que una persona llena un formulario del gobierno o una solicitud de crédito, envía un pedido por correo, solicita un empleo, expide un cheque o usa una tarjeta de crédito, se crea un rastro electrónico de datos. Cada vez que esos archivos son compartidos entre dependencias gubernamentales o empresas privadas, el individuo pierde algo de su privacidad.

Para ayudar a proteger los derechos constitucionales de los ciudadanos estadounidenses, se aprobó en 1970 la Ley de Información Crediticia Imparcial, seguida por la Ley Federal de Privacidad en 1974. Estas leyes especifican que es ilegal que un negocio conserve archivos secretos, que usted tiene el derecho de examinar y corregir los datos recolectados sobre usted, y que las dependencias gubernamentales y contratistas deben mostrar una justificación para tener acceso a sus registros. Continúan los esfuerzos por crear mecanismos que sirvan para preservar la seguridad y privacidad del individuo.

Para asegurar que los ejemplos presentados en este texto son compatibles con todos los sistemas operativos enlistados en la tabla 8.1, en general, pero no de manera exclusiva, nos apegamos a las especificaciones más restrictivas de DOS. Sin embargo, si usa uno de los otros sistemas operativos, deberá aprovechar las ventajas del aumento en la especificación de longitud para crear nombres de archivo descriptivos. Deberán evitarse los nombres de archivo largos debido a que requieren más tiempo para ser mecanografiados y pueden producir errores en esta tarea. Un largo manejable para un nombre de archivo es de 12 a 14 caracteres, con un máximo de 25 caracteres.

Usando la convención de DOS, todos los siguientes son nombres de archivo de datos de computadora válidos:

```
precios.dat     registro        info.txt
exper1.dat      calif.dat       fisica.mem
```

Elija nombres de archivo que indiquen el tipo de datos que hay en el archivo y la aplicación para la cual se usan. Con frecuencia, los primeros ocho caracteres describen los datos y una extensión (los caracteres después del punto decimal) describen la aplicación. Por ejemplo, el programa de hoja de cálculo Excel aplica de manera automática una extensión "xls" a todos los archivos de hoja de cálculo, los programas de procesamiento de palabras Word de Microsoft y WordPerfect usan las extensiones "doc" y "wp*x*" (donde *x* se refiere al número de la versión), respectivamente, y los compiladores de C++ requieren que un archivo de programa tenga la extensión "cpp". Cuando cree sus propios nombres de archivo, deberá observar esta práctica.

Usando la convención de DOS, el nombre "exper1.dat" es apropiado para describir un archivo de datos correspondiente al experimento número 1.

Punto de información

Flujos de entrada y salida

Un *flujo* es una ruta de transmisión en un solo sentido entre una fuente y un destino. Un flujo de bytes es enviado por esta ruta de transmisión. Una buena analogía para este flujo de bytes es un arroyo de agua que proporciona una ruta unidireccional para que el agua viaje de una fuente a un destino.

Los objetos de flujo se crean a partir de clases de flujo. Dos objetos de flujo que se han usado en forma extensa son el objeto de flujo de entrada llamado `cin` y el objeto de flujo de salida llamado `cout`. El objeto `cin` proporciona una ruta de transmisión del teclado al programa, y el objeto `cout` proporciona una ruta de transmisión del programa a la pantalla de la terminal. Estos dos objetos son creados a partir de las clases de flujo `istream` y `ostream`, respectivamente, las cuales son clases madre de la clase `iostream`. Cuando el archivo de encabezado `iostream` se incluye en un programa usando la directiva `#include <iostream>`, los objetos `cin` y `cout` son declarados y abiertos de manera automática por el compilador de C++ para el programa compilado.

Los objetos de flujo de archivos proporcionan las mismas capacidades que los objetos `cin` y `cout`, excepto que se conectan a un programa por medio de un archivo en lugar de al teclado o a la pantalla de la terminal. Los objetos de flujo de archivo deben ser declarados en forma explícita. Los objetos de flujo de archivo usados para entrada deben ser declarados como objetos de la clase `ifstream`. Los objetos de flujo de archivo usados para salida deben ser declarados como objetos de la clase `ofstream`. Las clases `ifstream` y `ofstream` se ponen a disposición de un programa por inclusión del archivo de encabezado `fstream`, usando la directiva `#include <fstream>`. La clase `fstream` se deriva de las clases `ifstream` y `ofstream` (véase la sección 8.8).

Existen dos tipos de **archivos básicos:** archivos de texto, los cuales se conocen como **archivos basados en caracteres,** y los **archivos basados en binarios**. Ambos tipos de archivos almacenan datos usando un código binario; la diferencia estriba en lo que representan los códigos. En resumen, los archivos basados en texto almacenan cada carácter individual, como una letra, dígito, signo de pesos, punto decimal, etc., usando un código de carácter individual (de manera típica ASCII o UNICODE). El uso de un código de caracteres permite a un procesador de palabras o a un editor de texto desplegar los archivos de modo que puedan ser leidos. Los archivos binarios usan el mismo código que su compilador C++ para sus tipos de datos primarios. Esto significa que los números aparecen en su forma binaria verdadera, mientras las cadenas conservan su forma ASCII o UNICODE. La ventaja de los archivos binarios es su compactibilidad debido a que se usa menos espacio para almacenar más números usando su código binario que como valores de carácter individuales. En general, la mayor parte de los archivos usados por los programadores son archivos de texto debido a que los datos del archivo pueden ser desplegados por programas de procesamiento de palabras y editores de texto simples. El tipo de archivo por omisión en C++ siempre es un archivo de texto y es el tipo de archivo que se presenta en este capítulo.

Objetos de flujo de archivos

Un **flujo de archivos** es una ruta de transmisión unidireccional utilizada para conectar un archivo almacenado en un dispositivo físico, como un disco o un CD-ROM, con un programa. Cada flujo de archivos tiene su propio método, el cual determina la dirección de los datos en la ruta de transmisión; es decir, si la ruta moverá datos de un archivo a un programa o si la ruta moverá datos de un programa a un archivo. Un flujo de archivos

que recibe o lee datos de un archivo a un programa se conoce como **flujo de archivos de entrada.** Un flujo de archivos que envía o escribe datos en un archivo se conoce como **flujo de archivos de salida.** La dirección, o modo, se define en relación con el programa y no al archivo; los datos que van a un programa se consideran datos de entrada, y los datos enviados desde el programa se consideran datos de salida. La figura 8.1 ilustra el flujo de datos desde y hacia un archivo usando flujos de entrada y de salida.

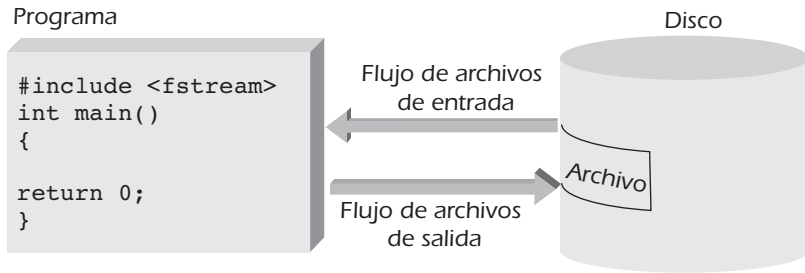


Figura 8.1 *Flujos de archivos de entrada y salida.*

Para cada archivo que utilice su programa, sin importar el tipo de archivo (texto o binario), debe crearse un objeto de flujo de archivos distinto. Si desea que su programa lea y escriba en un archivo, se requieren un objeto de flujo de archivos de entrada y uno de salida. Los objetos de flujo de archivos de entrada se declaran como tipo `ifstream`, y los flujos de archivos de salida como tipo `ofstream`. Por ejemplo, examine la siguiente instrucción de declaración:

```
ifstream archivo_entr;
```

Esta instrucción declara que un objeto de flujo de archivos de entrada llamado `archivo_entr` es un objeto de la clase `ifstream`. Del mismo modo, examine la siguiente instrucción de declaración:

```
ofstream archivo_sal;
```

Esta instrucción declara que un objeto de flujo de archivos de salida llamado `archivo_sal` es un objeto de la clase ofstream. Dentro de C++, se tiene acceso a un flujo de archivos por su nombre de objeto de flujo apropiado: un nombre para leer el archivo y un nombre para escribir en el archivo. Los nombres de objetos, como `archivo_entr` y `archivo_sal`, pueden ser cualquier nombre seleccionado por el programador que se ajuste a las reglas de identificación de C++.

Métodos de flujo de archivos

Cada objeto de flujo de archivos tiene acceso a los métodos definidos por su clase `ifstream` u `ofstream` respectiva. Estos métodos incluyen conectar un nombre de objeto de flujo a un nombre de archivo externo (llamado **abrir un archivo**), determinar si se ha hecho una conexión exitosa, cerrar una conexión (llamado **cerrar un archivo**), obtener el siguiente elemento de datos para el programa desde un flujo de entrada, colocar un elemento de datos nuevo del programa en un flujo de salida y detectar cuando se ha alcanzado el final de un archivo.

Abrir un archivo conecta cada objeto de flujo de archivos con su nombre de archivo externo específico. Esto se logra por medio de un método de apertura de flujo de archivos, el cual tiene dos propósitos. Primero, abrir un archivo que establece el vínculo de cone-

xión física entre un programa y un archivo. Dado que los detalles de este vínculo son manejados por el sistema operativo de la computadora y éstos son transparentes para el programa, por lo general el programador no necesita considerarlos.

Desde la perspectiva de codificación, el segundo propósito de abrir un archivo es más relevante. Además de establecer la conexión física real entre un programa y un archivo de datos, abrir un archivo conecta el nombre externo del archivo en la computadora con el nombre del objeto de flujo usado en forma interna por el programa. El método que realiza esta tarea se llama `open()` y es proporcionado por las clases `ifstream` y `ofstream`.

Al usar el método `open()` para conectar el nombre externo del archivo con su nombre de objeto de flujo interno, sólo se requiere un argumento, el cual es el nombre externo del archivo. Por ejemplo, examine la siguiente instrucción:

```
archivo_entr.open("precios.dat");
```

Conecta el archivo de texto externo llamado `precios.dat` con el objeto de flujo de archivo interno del programa llamado `archivo_entr`. Esto supone, por supuesto, que `archivo_entr` se ha declarado como un objeto `ifstream` u `ofstream`. Si se ha abierto un archivo con la instrucción anterior, el programa tiene acceso al archivo usando el nombre de objeto interno `archivo_entr`, y la computadora guarda el archivo bajo el nombre externo `precios.dat`. El argumento del nombre de archivo externo transmitido a `open()` es una cadena contenida entre comillas. Llamar al método `open()` requiere la notación de objeto estándar donde el nombre del método deseado, es este caso `open()`, es precedido por un punto y un nombre de objeto.

Cuando un archivo existente se conecta a un flujo de archivos de entrada, los datos del archivo quedan disponibles para entrada, empezando por el primer elemento de datos en el archivo. Del mismo modo, un archivo conectado a un flujo de archivos de salida crea un archivo nuevo y hace que el archivo quede disponible para salida. Si existe un archivo con el mismo nombre que el archivo abierto en modo de salida, el archivo antiguo es borrado y todos los datos se pierden.

Cuando se abre un archivo, para entrada o salida, la buena práctica de programación requiere que se compruebe la conexión que se ha establecido antes de intentar usar el archivo. Esto se puede hacer por medio del método `fail()`, el cual devolverá un valor verdadero si el archivo no fue abierto en forma exitosa (es decir, es verdadero en cuanto falló la apertura) o un valor `falso` si la apertura tuvo éxito. Por lo general, el método `fail()` se usa en un código similar al siguiente, el cual intenta abrir un archivo llamado `precios.dat` para entrada, comprueba que se hizo una conexión válida y reporta un mensaje de error si el archivo no fue abierto con éxito para entrada:

```
ifstream archivo_entr; // cualquier nombre de objeto puede
usarse aquí
archivo_entr.open("precios.dat"); // abre el archivo

// comprueba que la conexión se abrió con éxito
if (archivo_entr.fail())
{
  cout << "\nEl archivo no se abrió con éxito"
       << "\n Por favor compruebe que el archivo existe en
       realidad."
       << endl;
  exit(1);
}
```

Si el método `fail()` devuelve un valor booleano `verdadero`, lo cual indica que la apertura falló, el código despliega un mensaje. Además, la función `exit()` es llamada, lo cual es una solicitud para que el sistema operativo termine la ejecución del programa de inmediato. La función `exit()` requiere la inclusión de la función de encabezado `cstdlib` en cualquier programa que utilice esta función, y el argumento de un solo número entero de `exit()` es transmitido en forma directa al sistema operativo para una posible acción posterior o para inspección del usuario. A través del resto del texto se incluirá este tipo de verificación de errores siempre que se abre un archivo. (La sección 8.3 muestra cómo usar el manejo de excepciones para el mismo tipo de comprobación de errores.)

Además del método `fail()`, C++ proporciona otros tres métodos, mostrados en la tabla 8.2, que pueden utilizarse para detectar el estado de un archivo. El uso de estos métodos adicionales se presenta al final de la siguiente sección.

Tabla 8.2 Métodos de estado del archivo

Prototipo	Descripción
`fail()`	Devuelve un valor booleano `verdadero` si el archivo no se ha abierto con éxito; de lo contrario, devuelve un valor booleano `falso`.
`eof()`	Devuelve un valor booleano `verdadero` si se ha intentado leer más allá del final del archivo; de lo contrario, devuelve un valor booleano `falso`. El valor se vuelve verdadero sólo cuando se lee el primer carácter después del último carácter de archivo válido.
`good()`	Devuelve un valor booleano `verdadero` mientras el archivo está disponible para uso del programa. Devuelve un valor booleano `falso` si se ha intentado una lectura después del final del archivo. El valor se vuelve falso sólo cuando se lee el primer carácter después del último carácter de archivo válido.
`bad()`	Devuelve un valor booleano `verdadero` si se ha intentado una lectura después del final del archivo; de lo contrario, devuelve un valor `falso`. El valor se vuelve verdadero sólo cuando se lee el primer carácter después del último carácter de archivo válido.

El programa 8.1 ilustra las instrucciones requeridas para abrir un archivo de entrada, incluyendo una rutina de comprobación de errores para asegurar que se ha obtenido una apertura exitosa. Se dice que un archivo abierto para entrada está en **modo de lectura.**

Programa 8.1

```
#include <iostream>
#include <fstream>
#include <cstdlib>   // necesaria para exit()
using namespace std;

int main()
{
  ifstream archivo_entr;

  archivo_entr.open("precios.dat");  // abre el archivo con el
                                     // nombre externo precios.dat
  if (archivo_entr.fail())  // comprueba una apertura exitosa
  {
    cout << "\nEl archivo no fue abierto con éxito"
         << "\n Por favor compruebe que el archivo existe en realidad."
         << endl;
    exit(1);
  }

  cout << "\nEl archivo se ha abierto con éxito para lectura."
       << endl;

  // las instrucciones para leer datos de un archivo se colocarán aquí

  return 0;
}
```

Una muestra de la ejecución del programa 8.1 produjo la siguiente salida:

```
El archivo se ha abierto con exito para lectura.
```

Se requiere una comprobación diferente para archivos de salida porque, si existe un archivo que tenga el mismo nombre que el archivo que se va a abrir en modo de salida, el archivo existente es borrado y todos sus datos se pierden. Para evitar esto, el archivo es abierto primero en modo de entrada para ver si existe. Si existe, el usuario tiene la opción de permitir que sea sobrescrito cuando se abra más tarde en modo de salida. El código usado para lograr esto está resaltado en el programa 8.2.

Programa 8.2

```
#include <iostream>
#include <fstream>
#include <cstdlib> // necesario para exit()
using namespace std;

int main()
{
  ifstream archivo_entr;
  ofstream archivo_sal;

  archivo_entr.open("precios.dat"); // intenta abrir el archivo para entrada

  char respuesta;

  if (!archivo_entr.fail()) // si no falla, el archivo existe
  {
    cout << "Existe un archivo con el nombre precios.dat.\n"
         << "Desea continuar y sobrescribirlo\n"
         << " con los datos nuevos (si o no): ";
    cin >> respuesta;
    if (tolower(respuesta) == 'n')
    {
      cout << "El archivo existente no sera sobrescrito." << endl;
       exit(1); // termina la ejecucion del programa
    }
  }
  archivo_sal.open("precios.dat"); // ahora abre el archivo para escritura

  if (archivo_entr.fail()) // comprobar una apertura con exito
  {
    cout << "\nEl archivo no se abrió con éxito"
         << endl;
    exit(1);
  }

  cout << "El archivo se ha abierto con éxito para salida."
       << endl;

  // las instrucciones para escribir en el archivo se colocarán aquí

  return 0;
}
```

Punto de información

Uso de cadenas C como nombres de archivo

Si elige usar una cadena C para almacenar un nombre de archivo externo, debe estar enterado de las siguientes restricciones:

La longitud máxima de la cadena C debe ser especificada dentro de corchetes inmediatamente después de ser declarada. Por ejemplo, examine la siguiente declaración:

```
char nombre_archivo[21] = "precios.dat";
```

El número 21 limita el número de caracteres que pueden ser almacenada en la cadena C. El número entre corchetes (21) representa uno más que el número máximo de caracteres que pueden asignarse a la variable. Esto se debe a que el compilador agrega un carácter de fin de cadena para terminarla. Por tanto, El valor de la cadena `"precios.dat"`, el cual consiste de once caracteres, es almacenado en realidad como 12 caracteres. El carácter extra es un marcador de fin de cadena suministrado por el compilador. En nuestro ejemplo, el valor máximo de la cadena asignable al nombre de archivo de la variable de cadena es un valor de cadena consistente de 20 caracteres.

Las dos ejecuciones siguientes se hicieron con el programa 8.2:

```
Existe un archivo con el nombre precios.dat.
Desea continuar y sobrescribirlo
 con los datos nuevos (si o no): no
El archivo existente no sera sobrescrito.
```

y

```
Existe un archivo con el nombre precios.dat.
Desea continuar y sobrescribirlo
 con los datos nuevos (si o no): si
El archivo se ha abierto con éxito para salida.
```

Aunque los programas 8.1 y 8.2 pueden utilizarse para abrir un archivo existente para lectura y escritura, respectivamente, ambos programas carecen de instrucciones para llevar a cabo una lectura o escritura y cerrar el archivo. Estos temas se expondrán en breve. Antes de continuar, hay que anotar que es posible combinar la declaración de un objeto `ifstream` u `ofstream` y su instrucción de apertura asociada en una instrucción. Por ejemplo, observe las dos instrucciones siguientes en el programa 8.1:

```
ifstream archivo_entr;
archivo_entr.open("precios.dat");
```

Pueden combinarse en una sola instrucción:

```
ifstream archivo_entr("precios.dat");
```

Nombres de archivo incrustados e interactivos

Los programas 8.1 y 8.2 tienen dos problemas:

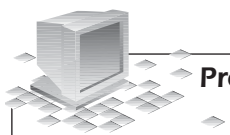
1. El nombre de archivo externo está incrustado dentro del código del programa.
2. No está previsto que un usuario introduzca el nombre de archivo deseado mientras se está ejecutando el programa.

Tal y como están escritos ambos programas, si se va a cambiar el nombre del archivo, el programador debe modificar el nombre de archivo externo en la llamada a `open()` y volver a compilar el programa. Ambos problemas pueden ser resueltos asignando el nombre de archivo a una variable de cadena.

Una variable de cadena como se ha usado a travéz del texto (véase el capítulo 9) es una variable que puede contener un valor de cadena, el cual es cualquier secuencia de cero o más caracteres encerrados entre comillas. Por ejemplo, `"Hola mundo"`, `"precios.dat"` y " " son cadenas. Hay que observar que las cadenas se escriben con comillas que delimitan el inicio y el final de una cadena pero no son almacenadas como parte de la cadena.

Al declarar e inicializar una variable de cadena para usarla en un método `open()`, la cadena es considerada como una cadena C. (Véase el Punto de información en la página anterior para las precauciones que deben tomarse cuando se use una cadena C.) Una alternativa más segura, y que se usará a tarvés de este texto, es usar un objeto de clase de cadena y convertir este objeto en una cadena C utilizando el método `c_str()`.

Una vez que se declara una variable de cadena para almacenar un nombre de archivo, puede utilizarse en una de dos formas. Primera, como se muestra en el programa 8.3a, puede colocarse en la parte superior de un programa para identificar con claridad el nombre externo de un archivo, en lugar de incrustarlo dentro de una llamada al método `open()`.



Programa 8.3a

```
#include <iostream>
#include <fstream>
#include <cstdlib>   // necesario para exit()
#include <string>
using namespace std;

int main()
{
  string nombre_archivo = "precios.dat"; // coloca el nombre de archivo al frente
  ifstream archivo_entr;

  archivo_entr.open(nombre_archivo.c_str()); // abre el archivo

  if (archivo_entr.fail()) // comprueba una apertura con exito
  {
    cout << "\nEl archivo llamado " << nombre_archivo << " no se abrió con éxito
         << "\n Por favor compruebe que el archivo existe en realidad."
         << endl;
    exit(1);
  }

  cout << "\nEl archivo se ha abierto con éxito para lectura.\n";

  return 0;
}
```

El programa 8.3a muestra que se ha declarado e inicializado el objeto de cadena con el siguiente nombre:

```
nombre_archivo
```

Se usa este nombre al principio de `main()` para una fácil identificación del archivo. Cuando se usa un objeto de cadena, en oposición a una literal de cadena, el nombre de la variable no se encierra entre comillas en la llamada al método `open()`. Dentro de la llamada `open()`, el objeto de cadena es convertido en una cadena C usando la siguiente expresión:

```
nombre_archivo.c_str().
```

Por último, en el código del método `fail()` el nombre externo del archivo es desplegado con facilidad al insertar el nombre del objeto de cadena en el flujo de salida `cout`. Por estas razones, se continuará identificando los nombres externos de los archivos de esta manera.

Otro papel útil que desempeñan los objetos de cadena es permitir al usuario introducir el nombre del archivo mientras se está ejecutando el programa. Por ejemplo, el código:

```
string nombre_archivo;

cout << "Por favor introduzca el nombre del archivo que
desea abrir: ";
cin >> nombre_archivo;
```

permite a un usuario introducir un nombre externo de archivo en tiempo de ejecución. La única restricción en este código es que el usuario no debe encerrar el valor de cadena introducido entre comillas, y el valor de cadena introducido no puede contener ningún espacio en blanco. La razón para ello que cuando se usa `cin`, el compilador terminará la cadena cuando encuentre un espacio en blanco. El programa 8.3b usa este código en el contexto de un programa completo.

La siguiente es una salida de muestra proporcionada por el programa 8.3b:

```
Por favor introduzca el nombre del archivo que desea
abrir: foobar

El archivo llamado foobar no se abrió con éxito
 Por favor compruebe que el archivo existe en realidad.
```

Cerrar un archivo

Un archivo se cierra usando el método `close()`. Este método rompe la conexión entre el nombre externo del archivo y el objeto de flujo de archivos, el cual puede ser usado por otro archivo. Examine la siguiente instrucción:

```
archivo_entr.close();
```

Esta instrucción cierra la conexión del flujo `archivo_entr` con su archivo actual. Como se indica, el método `close()` no utiliza un argumento.

Debido a que todas las computadoras tienen un límite en el número máximo de archivos que pueden abrirse a la vez, cerrar archivos que ya no son necesarios tiene sentido. Los archivos abiertos existentes al final de la ejecución del programa normal serán cerrados de manera automática por el sistema operativo.

Programa 8.3b

```
#include <iostream>
#include <fstream>
#include <cstdlib>  // necesario para exit()
#include <string>
using namespace std;
int main()
{
  string nombre_archivo;
  ifstream archivo_entr;

  cout << "Por favor introduzca el nombre del archivo que desea abrir: ";
  cin >> nombre_archivo;

  archivo_entr.open(nombre_archivo.c_str());  // abre el archivo

  if (archivo_entr.fail())  // comprueba una apertura con éxito
  {
    cout << "\nEl archivo llamado " << nombre_archivo << " no se abrio con exito
         << "\n Por favor compruebe que el archivo existe en realidad."
         << endl;
    exit(1);
  }
  cout << "\nEl archivo se abrió con éxito para lectura.\n";

  return 0;
}
```

Punto de información

Uso de objetos `fstream`

Al usar objetos `ifstream` y `ofstream`, el modo de entrada o salida está implicado por el objeto. Por tanto, los objetos `ifstream` deben usarse para entrada y los objetos `ofstream` deben utilizarse para salida.

Otro medio para crear flujos de archivos es usar objetos `fstream` que puedan utilizarse para entrada o salida, pero este método requiere una designación explícita. Un objeto `fstream` se declara usando la siguiente sintaxis:

```
fstream nombreObjeto;
```

Cuando se usa el método `open()` de la clase `fstream()`, se requieren dos argumentos: un nombre externo de archivo y un indicador de modo. Aquí se muestran los indicadores de modo permisibles:

Indicador	Descripción
`ios::in`	Abre un archivo de texto en modo de entrada
`ios::out`	Abre un archivo de texto en modo de salida
`ios::app`	Abre un archivo de texto en modo anexar
`ios::ate`	Va al final del archivo abierto
`ios::binary`	Abre un archivo binario en modo de entrada (es archivo de texto el valor por omisión)
`ios::trunc`	Elimina el contenido del archivo si existe
`ios::nocreate`	Si el archivo no existe, la apertura falla
`ios::noreplace`	Si el archivo existe, falla la apertura para salida

Como con los objetos `ofstream`, un objeto `fstream` en modo de salida crea un archivo nuevo y hace que el archivo esté disponible para escritura. Si existe un archivo con el mismo nombre que el archivo abierto para salida, el archivo antiguo es borrado. Por ejemplo, suponga que se ha declarado `archivo1` como un objeto de tipo `fstream` usando la siguiente instrucción:

```
fstream archivo1;
```

La instrucción que sigue intenta abrir el archivo de texto nombrado `precios.dat` para salida:

```
archivo1.open("precios.dat"),ios::out);
```

Una vez que se ha abierto este archivo, el programa tiene acceso al archivo usando el nombre de objeto interno `archivo1`, y la computadora guarda el archivo bajo el nombre externo `precios.dat`.

Un objeto de archivo `fstream` abierto en modo anexar significa que un archivo existente está disponible para que se anexen datos al final del archivo. Si el archivo abierto para anexar no existe, se crea un archivo nuevo con el nombre designado y se hace disponible para recibir salida del programa. Por ejemplo, suponga que se ha declarado que `archivo1` es del tipo `fstream`:

```
archivo1.open("precios.dat",ios::app);
```

La instrucción anterior intenta abrir un archivo de texto llamado `precios.dat` y lo hace disponible para que se anexen datos al final del archivo.

Por último, un objeto `fstream` abierto en modo de entrada significa que se ha conectado un archivo externo existente y que sus datos están disponibles como entrada. Por ejemplo, suponga que se ha declarado que `archivo1` es del tipo `fstream`

```
archivo1.open("precios.dat",ios::in);
```

(Continúa)

(Continuación)

La instrucción anterior intenta abrir un archivo de texto llamado `precios.dat` para entrada. Los indicadores de modo pueden combinarse con la operación de bit O (véase la sección 15.2).

```
archivo1.open("precios.dat", ios::in | ios::binary)
```

La instrucción anterior abre el flujo `archivo1`, el cual puede ser un `fstream` o `ifstream`, como un flujo binario de entrada. Si se omite el indicador de modo como el segundo argumento para un objeto `ifstream`, el flujo es abierto, por omisión, como un archivo de entrada de texto; si se omite el indicador de modo para un objeto `ofstream`, el flujo es abierto, por omisión, como un archivo de salida de texto.

Ejercicios 8.1

1. Escriba instrucciones de declaración y apertura individuales que vinculen los siguientes nombres de archivos de datos externos con sus correspondientes nombres de objeto internos. Suponga que todos los archivos se basan en texto.

Nombre externo	Nombre de objeto	Modo
`coba.mem`	`memo`	`salida`
`libro.let`	`carta`	`salida`
`cupones.bnd`	`cupones`	`anexar`
`produce.bnd`	`produce`	`anexar`
`precios.dat`	`archivo_precios`	`entrada`
`indices.dat`	`indices`	`entrada`

2. a. Escriba un conjunto de dos instrucciones que declare primero los siguientes objetos como objetos `ifstream` y luego los abra como archivos de entrada de texto: `enDatos.txt`, `precios.txt`, `cupones.dat` y `exper.dat`.
b. Vuelva a escribir las dos instrucciones para el ejercicio 2a usando una sola instrucción.

3. a. Escriba un conjunto de dos instrucciones que declaren primero los siguientes objetos como objetos `ofstream` y luego los abra como archivos de salida de texto: `salFecha.txt`, `indices.txt`, `trecho.txt` y `archivo2.txt`.
b. Vuelva a escribir las dos instrucciones para el ejercicio 3a usando una sola instrucción.

4. Introduzca y ejecute el programa 8.1 en su computadora.

5. Introduzca y ejecute el programa 8.2 en su computadora.

6. a. Introduzca y ejecute el programa 8.3a en su computadora.
b. Agregue un método `close()` al programa 8.3a y luego ejecute el programa.

7. a. Introduzca y ejecute el programa 8.3b en su computadora.
b. Agregue un método `close()` al programa 8.3b y luego ejecute el programa.

Punto de información

Comprobar una conexión exitosa

Debe comprobarse que el método `open()` estableció con éxito una conexión entre un flujo de archivo y un archivo externo. Esto se debe a que la llamada `open()` es una solicitud al sistema operativo que puede fallar por varias razones. La principal de estas razones puede ser una solicitud para abrir un archivo existente para lectura que el sistema operativo no puede localizar o una solicitud para abrir un archivo para salida en una carpeta inexistente. Si el sistema operativo no puede satisfacer la solicitud de apertura, usted necesita saberlo y terminar el programa. No hacerlo así puede producir un comportamiento anormal del programa o una caída subsiguiente de éste.

Hay dos estilos de codificación para comprobar el valor devuelto. El método más común para comprobar que no ocurrió una falla cuando se intenta usar un archivo de entrada es el codificado en el programa 8.1. Se usa para distinguir la solicitud `open()` de la comprobación hecha por medio de la llamada `fail()` y a continuación se repite por comodidad:

```
archivo_entr.open("precios.dat"); // solicitud para abrir
el archivo

if (archivo_entr.fail()) // comprueba una conexión fallida
{
  cout << "\nEl archivo no se abrió con éxito"
       << "\n Por favor compruebe que el archivo existe
          en realidad."
       << endl;
  exit(1);
}
```

Del mismo modo, la comprobación hecha en el programa 8.2 se incluye por lo general cuando se está abriendo un archivo en modo de salida.

De manera alternativa, pueden encontrarse programas que usan objetos `fstream` en lugar de objetos `ifstream` y `ofstream` (véase el recuadro Punto de información anterior). Cuando se usa el método `open()` de `fstream` se requieren dos argumentos: un nombre externo del archivo y una indicación explícita del modo. Usando un objeto `fstream` la solicitud para abrir y comprobar un archivo de entrada por lo general aparece como sigue:

```
fstream archivo_entr;

archivo_entr.open("nombre de archivo externo", ios::in);
if (archivo_entr.fail())
{
  cout << "\nEl archivo no se abrió con éxito"
       << "\n Por favor compruebe que el archivo existe
          en realidad."
       << endl;
  exit(1);
}
```

Muchas veces la expresión condicional `archivo_entr.fail()` es reemplazada por la expresión equivalente `!archivo_entr`. Aunque nosotros siempre usamos objetos `ifstream` y `ofstream`, debe estar preparado para encontrar los estilos que usan objetos `fstream`.

8. Usando los manuales de referencia proporcionados con el sistema operativo de su computadora, determine lo siguiente:
   a. el número máximo de caracteres que pueden usarse para nombrar un archivo para almacenamiento en el sistema de su computadora
   b. el número máximo de archivos de datos que pueden abrirse al mismo tiempo
9. ¿Sería apropiado llamar archivo a un programa en C++ guardado? ¿Por qué sí o por qué no?
10. a. Escriba instrucciones de declaración y apertura individuales para vincular los siguientes nombres de archivo de datos externos con sus nombres de objeto internos correspondientes. Use sólo objetos `ifstream` y `ofstream`.

Nombre externo	Nombre de objeto	Modo
`coba.mem`	`memo`	`binario y salida`
`cupones.bnd`	`cupones`	`binario y anexar`
`precios.dat`	`archivo_precios`	`binario y entrada`

   b. Vuelva a hacer el ejercicio 10a usando sólo objetos `fstream`.
   c. Escriba instrucciones de cierre para cada uno de los archivos abiertos en el ejercicio 10a.

8.2 LECTURA Y ESCRITURA DE ARCHIVOS BASADOS EN CARACTERES

Leer o escribir en archivos basados en caracteres implica las operaciones casi idénticas para leer la entrada realizada desde un teclado y escribir datos a una pantalla de despliegue. Para escribir en un archivo, el objeto `cout` es reemplazado por el nombre de objeto `ofstream` declarado en el programa. Por ejemplo, si `archivo_sal` es declarado como un objeto de tipo `ofstream`, las siguientes instrucciones de salida son válidas:

```
archivo_sal << 'a';
archivo_sal << "¡Hola mundo!";
archivo_sal << descrip << ' ' << precio;
```

El nombre de archivo en cada una de estas instrucciones, en lugar de `cout`, dirige el flujo de salida a un archivo específico en lugar de al dispositivo de despliegue estándar. El programa 8.4 ilustra el uso del operador de inserción, `<<`, para escribir una lista de descripciones y precios en un archivo.

Programa 8.4

```
#include <iostream>
#include <fstream>
#include <cstdlib>   // necesario para exit()
#include <string>
#include <iomanip>   // necesario para formatear
using namespace std;

int main()
{
  string nombre_archivo = "precios.dat"; // pone al frente el nombre de archivo
  ofstream archivo_sal;

  archivo_sal.open(nombre_archivo.c_str());

  if (archivo_sal.fail())
  {
    cout << "El archivo no se abrió con éxito" << endl;
    exit(1);
  }

  // establece los formatos del flujo de archivo de salida
  archivo_sal << setiosflags(ios::fixed)
              << setiosflags(ios::showpoint)
              << setprecision(2);

  // envía datos al archivo
  archivo_sal << "Alfombras " << 39.95 << endl
              << "Bombillas " << 3.22 << endl
              << "Fusibles " << 1.08 << endl;

  archivo_sal.close();
  cout << "El archivo " << nombre_archivo
       << " se ha escrito con éxito." << endl;

  return 0;
}
```

Punto de información

Formatear los datos del flujo de salida de un archivo de texto

Los flujos de archivo de salida pueden formatearse de la misma manera que el flujo de salida cout estándar. Por ejemplo, si se ha declarado un flujo de salida nombrado `salidaArchivo`, la siguiente instrucción formatea todos los datos insertados en el flujo `salidaArchivo` en la misma forma en que trabajan estos manipuladores parametrizados para el flujo `cout`:

```
salidaArchivo << setiosflags(ios::fixed)
              << setiosflags(ios::showpoint)
              << setprecision(2);
```

El primer parámetro manipulador, `ios::fixed`, causa que el flujo dé salida a todos los números como si fueran valores de punto flotante. El siguiente parámetro, `ios::showpoint`, le indica al flujo que proporcione un punto decimal. Por tanto, un valor como 1.0 aparecerá como 1.0, y no 1. Por último, el manipulador `setprecision` le indica al flujo que despliegue dos valores decimales después del punto decimal. Por tanto, el número 1.0, por ejemplo, aparecerá como 1.00.

En lugar de manipuladores, también pueden usarse los métodos de flujo `setf()` y `precision()`. Por ejemplo, el formato anterior puede elaborarse usando el siguiente código:

```
salidaArchivo.setf(ios::fixed);
salidaArchivo.setf(ios::showpoint);
salidaArchivo.precision(2);
```

El estilo que se elija es cuestión de preferencia. En ambos casos, sólo es necesario especificar los formatos una vez, y permanecen en efecto para todos los números insertados después en el flujo de archivo.

Cuando se ejecuta el programa 8.4, se crea un archivo, `precios.dat`, y es guardado por la computadora como un archivo de texto. El archivo es un archivo secuencial consistente en los siguientes datos:

```
Alfombras 39.95
Bombillas 3.22
Fusibles 1.08
```

El almacenamiento real de caracteres en el archivo depende de los códigos de carácter usados por la computadora. Aunque sólo parecen estar almacenados 42 caracteres en el archivo, correspondientes a las descripciones, espacios en blanco y precios escritos en el archivo, el archivo contiene 48 caracteres.

Los caracteres extra consisten en la secuencia de escape de línea nueva al final de cada línea creada por el manipulador `endl`, el cual es creado como un carácter de retorno de carro (`cr`) y avance de línea (`lf`). Suponiendo que los caracteres son almacenados usando el código ASCII, el archivo `precios.dat` está almacenado en forma física como se ilustra en la figura 8.2. Por comodidad, el carácter correspondiente a cada código hexadecimal se enlista abajo del código. Un código de 20 representa el carácter de espacio en blanco. Además, C y C++ anexan el byte hexadecimal de valor bajo 0x00 como el centinela de fin de archivo (EOF) cuando se cierra el archivo. Este centinela EOF nunca se cuenta como parte del archivo.

Punto de información

El método `put()`

Todos los flujos de salida tienen acceso al método `put()` de la clase `fstream`, el cual permite la salida carácter por carácter a un flujo. Este método funciona de la misma manera que el operador de inserción de carácter, <<. La sintaxis de esta llamada al método es la siguiente:

```
ofstreamNombre.put(expresión-en-carácter);
```

La *`expresión-en-carácter`* puede ser una variable de carácter o un valor literal. Por ejemplo, el siguiente código puede usarse para dar salida a una 'a' al flujo de salida estándar:

```
cin.put('a');
```

De una manera similar, si `archivo_sal` es un archivo de objeto `ofstream` que se ha abierto, el siguiente código envía el valor de carácter en la variable de carácter llamada `codigoclave` a esta salida:

```
char codigoclave
  .
  .
archivo_sal.put(codigoclave);
```

```
46 75 73 69 62 6C 65 73 20 33 39 2E 39 35 0D 0A 42 6F 6D 62 69 6C 6C 61 73 20
 F  u  s  i  b  l  e  s     3  9  .  9  5 cr 1f  B  o  m  b  i  l  l  a  s

33 2E 32 32 0D 0A 41 6C 66 6F 6D 62 72 61 73 20 31 2E 30 38 0D 0A
 3  .  2  2 cr 1f  A  l  f  o  m  b  r  a  s     1  .  0  8 cr 1f
```

Figura 8.2 *El archivo* `precios.dat` *tal como es almacenado por la computadora.*

Lectura de un archivo de texto

Leer datos de un archivo basado en texto es casi igual a leer datos de un teclado estándar, excepto que el objeto `cin` es reemplazado por el objeto `ifstream` declarado en el programa. Por ejemplo, si `archivo_entr` es declarado como un objeto de tipo `ifstream` que es abierto para entrada, la entrada que sigue a la instrucción leerá los siguientes dos elementos en el archivo y los almacenará en las variables `descrip` y `precio`:

```
archivo_entr >> descrip >> precio;
```

El nombre del flujo de archivos en esta instrucción, en lugar de `cin`, instruye que la entrada provenga del flujo de archivo en lugar de venir del flujo del dispositivo de entrada estándar. Otros métodos que pueden usarse para flujo de entrada se muestran en la tabla 8.3. Cada uno de estos métodos debe ser precedido por un nombre de objeto de flujo.

Tabla 8.3 Métodos `fstream`

Nombre del método	Descripción
`get()`	Devuelve el siguiente carácter extraído del flujo de entrada como un `int`.
`get(charVar)`	Versión sobrecargada de `get()` que extrae el siguiente carácter del flujo de entrada y lo asigna a la variable de carácter especificada, `charVar`.
`getline( strObj, termChar)`	Extrae caracteres del flujo de entrada especificado, `strObj`, hasta que se encuentra el carácter de terminación, `termChar`. Asigna los caracteres al objeto de clase de cadena especificado, `strObj`.
`peek()`	Devuelve el siguiente carácter en el flujo de entrada sin extraerlo del flujo.
`ignore(int n)`	Se salta los siguientes *n* caracteres. Si se omite `n`, el valor por omisión es saltarse el siguiente carácter individual.

El programa 8.5 ilustra cómo puede leerse el archivo `precios.dat` que fue creado en el programa 8.4. El programa ilustra un método para detectar el marcador EOF usando la función `good()` (véase la tabla 8.2). Debido a que esta función devuelve un valor booleano verdadero antes que se haya leído o transmitido el marcador EOF, puede utilizarse para verificar que los datos leídos son datos de archivo válidos. Sólo después que se ha leído o transmitido el marcador EOF esta función devuelve un valor booleano `falso`. Por tanto, la notación `while(archivo_entr.good())` usada en el programa 8.5 asegura que los datos pertenecen al archivo antes que se haya leído EOF.

El despliegue producido por el programa 8.5 es el siguiente:

```
Alfombras 39.95
Bombillas 3.22
Fusibles 1.08
```

Examine la expresión `archivo_entr.good()` usada en la instrucción while. Esta expresión es verdadera en tanto el marcador EOF no haya sido leído. Por tanto, mientras la lectura del elemento fue buena, el ciclo continúa leyendo el archivo. Dentro del ciclo, los elementos que se acaban de leer son desplegados, y luego una cadena nueva y un número de precisión doble son introducidos al programa. Cuando se ha detectado el EOF, la expresión devuelve un valor booleano de `falso` y el ciclo termina. Esto asegura que los datos son leídos y desplegados hasta el marcador EOF, pero sin incluirlo.

Un reemplazo directo para la instrucción `while(archivo_entr.good())` es la instrucción `while(!archivo_entr.eof())`, la cual se lee "mientras el final del archivo no se ha alcanzado". Esto funciona porque la función `eof()` devuelve un valor booleano verdadero sólo después que se ha leído o transmitido el marcador EOF. En efecto, la expresión relacional comprueba que el EOF no ha sido leído; de ahí el uso del operador NOT, `!`.

Otro medio de detectar el EOF es usar el hecho que la operación de extracción, `>>`, devuelve un valor booleano `verdadero` si los datos son extraídos de un flujo; de lo contrario, devuelve un valor booleano `falso`. Usando este valor devuelto, puede utilizarse el siguiente código dentro del programa 8.5 para leer el archivo.

Programa 8.5

```
#include <iostream>
#include <fstream>
#include <cstdlib>   // necesario para exit()
#include <string>
using namespace std;

int main()
{
  string nombre_archivo = "precios.dat"; // pone al frente el nombre de archivo
  string descrip;
  double precio;

  ifstream archivo_entr;

  archivo_entr.open(nombre_archivo.c_str());

  if (archivo_entr.fail()) // comprueba el éxito en la apertura
  {
    cout << "\nEl archivo no se abrió con éxito"
         << "\n Por favor compruebe que el archivo existe en realidad."
         << endl;
    exit(1);
  }

  // lee y despliega el contenido del archivo
  archivo_entr >> descrip >> precio;
  while (archivo_entr.good()) // comprueba el siguiente carácter
  {
    cout << descrip << ' ' << precio << endl;
    archivo_entr >> descrip >> precio;archivo_entr >> descrip >> precio;
  }

  inFile.close();

  archivo_entr.close();
}
```

Punto de información

Una forma de identificar el nombre y ubicación de un archivo

Durante el desarrollo del programa, los archivos de prueba se colocan por lo general en el mismo directorio que el programa. Por consiguiente, una llamada al método como `archivo_entr.open ("exper.dat")` no causa problemas al sistema operativo. En sistemas de producción, sin embargo, no es poco común que los archivos de datos residan en un directorio mientras los archivos del programa residen en otro. Por esta razón siempre es una buena idea incluir el nombre de la ruta completa de cualquier archivo abierto.

Por ejemplo, si el archivo `exper.dat` reside en el directorio `C:\prueba\archivos`, la llamada `open()` deberá incluir el nombre de la ruta completa: `archivo_entr.open("C:\\prueba \\archivos\\exper.dat")`. Entonces, sin importar desde dónde se ejecute el programa, el sistema operativo sabrá dónde localizar el archivo. Observe el uso de diagonales invertidas dobles, lo cual es necesario.

Otra convención importante es enlistar todos los nombres de archivo al principio del programa en lugar de incrustar los nombres en las profundidades dentro del código. Esto puede lograrse con facilidad con variables de cadena para almacenar cada nombre de archivo.

Por ejemplo, si las instrucciones:

```
string nombre_archivo = "c:\\prueba\\archivos\\exper.dat";
```

se colocan al principio de un archivo de programa, la instrucción de declaración enlista con claridad tanto el nombre del archivo deseado como su ubicación. Luego, si se va a probar algún otro archivo, todo lo que se requiere es un cambio simple de una línea al principio del programa.

Usar una variable de cadena para el nombre del archivo también es útil para la comprobación del método `fail()`. Por ejemplo, considere el siguiente código:

```
string nombre_archivo;
ifstream archivo_entr;

archivo_entr.open(nombre_archivo.c_str());

if (archivo_entr.fail())
{
  cout << "\n El archivo nombrado " << nombre_archivo
       << " no se abrió con éxito"
       << "\n Por favor compruebe que este archivo existe
          en realidad."
       exit(1);
}
```

En este código, el nombre del archivo que falló en abrirse es desplegado en forma directa dentro del mensaje de error sin que el nombre esté incrustado como un valor de cadena.

```
// leer y desplegar el contenido del archivo
  while (archivo_entr >> descrip >> precio) // comprueba
  el siguiente carácter
    cout << descrip << ' ' << precio << endl;
```

Aunque al principio es un poco críptico, este código tiene sentido perfecto cuando se entiende que la expresión que se está probando extrae datos del archivo y devuelve un valor booleano para indicar si la extracción fue exitosa.

Por último, en la instrucción `while` previa o en el programa 8.5, la expresión `archivo_entr >> descrip >>` precio puede reemplazarse con un método `getline()` (véase la sección 7.2). Para entrada de archivo, este método tiene la siguiente sintaxis:

```
getline(objetoArchivo, strObj, carácter-de-terminación)
```

objetoArchivo es el nombre del archivo *ifstream*, *strObj* es un objeto de la clase de cadena y *carácter-de-terminación* es una constante o variable de carácter opcional que especifica el carácter de terminación. Si se omite este tercer argumento opcional, el carácter de terminación por omisión es el carácter de línea nueva (`'\n'`). El programa 8.6 ilustra el uso de *getline()* dentro del contexto de un programa completo.

Programa 8.6

```
#include <iostream>
#include <fstream>
#include <cstdlib>   // necesario para exit()
#include <string>
using namespace std;

int main()
{
  string nombre_archivo = "precios.dat"; // pone al frente el nombre de archivo
  string linea;
  ifstream archivo_entr;

  archivo_entr.open(nombre_archivo.c_str());

  if (archivo_entr.fail()) // comprueba que se abrió con éxito
  {
    cout << "\nEl archivo no se abrió con éxito"
         << "\n Por favor compruebe que el archivo existe en realidad."
         << endl;
    exit(1);
  }

  // lee y despliega el contenido del archivo
  while (getline(archivo_entr,linea))
    cout << linea << endl;

  archivo_entr.close();

  return 0;
}
```

Punto de información

Los métodos `get()` y `putback()`

Todos los flujos de entrada tienen acceso al método `get()` de la clase fstream, el cual permite la entrada carácter por carácter desde un flujo de entrada. Este método funciona de manera similar a la extracción de carácter usando el operador >>, con dos diferencias importantes: si se encuentra un carácter de línea nueva, `'\n'`, o un carácter de espacio en blanco, `' '`, cada uno de estos caracteres es leído de la misma manera que cualquier otro carácter alfanumérico. La sintaxis de la llamada a este método es la siguiente:

```
istreamNombre.get(variable-de-carácter);
```

Por ejemplo, puede usarse el siguiente código para leer el siguiente carácter del flujo de entrada estándar y almacenar el carácter en la variable `ch`:

```
char ch;
cin.get(ch);
```

De manera similar, si `archivo_entr` es un objeto `ifstream` que se ha abierto a un archivo, el siguiente código lee el siguiente carácter en el flujo y lo asigna al carácter `codigoclave`:

```
char codigoclave;
archivo_entr.get(codigoclave);
```

Además del método `get()`, todos los flujos de entrada tienen un método `putback()` que puede utilizarse para poner el último carácter leído de un flujo de entrada de vuelta en el flujo. Este método tiene la siguiente sintaxis:

```
ifstreamNombre.putback(expresión-de-carácter);
```

`expresión-de-carácter` puede ser cualquier variable de carácter o valor de carácter.

El método `putback()` proporciona una capacidad de salida a un flujo de entrada. El carácter `putback` no necesita ser el último carácter leído; más bien, puede ser cualquier carácter. Sin embargo, todos los caracteres de `putback` no tienen efecto en el archivo de datos sino sólo en el flujo de entrada abierto. Por tanto, los caracteres del archivo de datos permanecen sin cambios aunque puedan cambiar los caracteres leídos en lo subsiguiente desde el flujo de entrada.

El programa 8.6 es un programa de copiado de texto línea por línea, el cual lee una línea de texto desde el archivo y luego la despliega en la terminal. La salida del programa 8.6 es la siguiente:

```
Alfombras 39.95
Bombillas 3.22
Fusibles 1.08
```

Si fuera necesario obtener la descripción y el precio como variables individuales, debería utilizarse el programa 8.5 o procesarse más la cadena devuelta por `getline()` en el programa 8.6 para extraer los elementos de datos individuales. (Véase la sección 8.7 para procedimientos de análisis sintáctico.)

Archivos estándar en dispositivos

Los objetos de flujo de archivos que se han usado han sido objetos de archivo lógicos. Un objeto de archivo lógico es un flujo que conecta un archivo de datos relacionados lógicamente, como un archivo de datos, con un programa. Además de objetos de archivo lógicos, C++ soporta objetos de archivo físicos. Un objeto de archivo físico es un flujo que conecta a un dispositivo de hardware, como un teclado, pantalla o impresora.

El dispositivo físico asignado a su programa para la introducción de datos se llama de manera formal **archivo estándar de entrada**. Por lo general, éste es el teclado. Cuando se encuentra una llamada al método de objeto `cin` en un programa en C++, es una solicitud al sistema operativo para que vaya a este archivo de entrada estándar por la entrada esperada. Del mismo modo, cuando se encuentra una llamada al método de objeto `cout`, la salida se despliega de manera automática o se "escribe" en un dispositivo que ha sido asignado como el **archivo estándar de salida**. Para la mayor parte de los sistemas, éste es una pantalla de computadora, aunque puede ser una impresora.

Cuando se ejecuta un programa, el flujo de entrada estándar `cin` es conectado con el dispositivo estándar de entrada. Del mismo modo, el flujo de salida estándar `cout` es conectado con el dispositivo estándar de salida. Estos dos flujos de objetos están disponibles para uso del programador, como lo están el flujo de error estándar, `cerr`, y el flujo de registro estándar, `clog`. Estos dos flujos se conectan con la pantalla de la terminal.

Otros dispositivos

Los flujos de teclado, despliegue, reporte de errores y registro son conectados en forma automática con los objetos de flujo llamados `cin`, `cout`, `cerr` y `clog`, respectivamente, cuando se incluye el archivo de encabezado `iostream` en un programa. Pueden utilizarse otros dispositivos para entrada o salida si se conoce el nombre asignado por el sistema. Por ejemplo, la mayor parte de las computadoras personales IBM o compatibles con IBM asignan el nombre `prn` a la impresora conectada a la computadora. Para estas computadoras, una instrucción como `archivo_sal.open("prn")` conecta la impresora al objeto `ofstream` llamado `archivo_sal`. Una instrucción subsiguiente, como `archivo_sal << "¡Hola mundo!";` causaría que la cadena `¡Hola mundo!` saliera en forma directa a la impresora. Como el nombre de un archivo real, `prn` debe encerrarse entre comillas en la llamada a la función `open()`.

Ejercicios 8.2

1. **a.** Introduzca y ejecute el programa 8.5.
   b. Modifique el programa 8.5 para usar la expresión `!archivo_entr.eof()` en lugar de la expresión `archivo_entr.good()`, y ejecute el programa para ver que opera en forma correcta.

2. **a.** Introduzca y ejecute el programa 8.6.
   b. Modifique el programa 8.6 reemplazando el identificador `cout` con `cerr`, y verifique que la salida para el flujo de archivo de error estándar es la pantalla.
   c. Modifique el programa 8.6 reemplazando el identificador cout con `clog`, y verifique que la salida para el flujo de registro estándar es la pantalla.

3. a. Escriba un programa en C++ que acepte líneas de texto del teclado y escriba cada línea en un archivo llamado `texto.dat` hasta que se introduzca una línea vacía. Una línea vacía es una línea sin texto que se crea oprimiendo la tecla Entrar (o Retorno).

b. Modifique el programa 8.6 para leer y desplegar los datos almacenados en el archivo `texto.dat` creado en el ejercicio 3a.

4. Determine el comando o procedimiento del sistema operativo proporcionado por su computadora para desplegar el contenido de un archivo guardado.

5. a. Cree un archivo de texto nombrado `empleado.dat` que contenga los siguientes datos:

Anthony	A	10031	7.82	12/18/05
Burrows	W	10067	9.14	6/ 9/04
Fain	B	10083	8.79	5/18/04
Janney	P	10095	10.57	9/28/04
Smith	G	10105	8.50	12/20/03

b. Escriba un programa en C++ para leer el archivo `empleado.dat` creado en el ejercicio 5a y producir una copia duplicada del archivo llamada `empleado.bak`.

c. Modifique el programa escrito en el ejercicio 5b para que acepte los nombres de los archivos original y duplicado como entrada del usuario.

d. El programa escrito para el ejercicio 5c siempre copia datos de un archivo original a un archivo duplicado. ¿Cuál es un mejor método para aceptar los nombres de archivo original y duplicado, otro que no sea indicar al usuario que los introduzca cada vez que se ejecute el programa?

6. a. Escriba un programa en C++ que abra un archivo y despliegue el contenido del archivo con números de línea asociados. Es decir, el programa deberá imprimir el número 1 antes de desplegar la primera línea, imprimir el número 2 antes de desplegar la segunda línea, y así en forma sucesiva para cada línea en el archivo.

b. Modifique el programa escrito en el ejercicio 6a para enlistar el contenido del archivo en la impresora asignada a su computadora.

7. a. Cree un archivo de texto que contenga los siguientes datos (sin los encabezados):

Nombres	Número de seguro social	Tarifa por hora	Horas trabajadas
B Caldwell	555-88-2222	7.32	37
D Memcheck	555-77-4444	8.32	40
R Potter	555-77-6666	6.54	40
W Rosen	555-99-8888	9.80	35

b. Escriba un programa en C++ que lea el archivo de datos creado en el ejercicio 7a y calcule y despliegue una lista de nómina. La salida deberá enlistar el número de seguro social, nombre, pago bruto para cada individuo y donde el pago bruto se calcula se hará como *tarifa por hora x horas trabajadas*.

8. a. Cree un archivo de texto que contenga los siguientes números de automóviles, número de millas recorridas y número de galones de gasolina utilizados en cada automóvil (no incluya los encabezados):

Núm. de automóvil	Millas recorridas	Galones utilizados
54	250	19
62	525	38
71	123	6
85	1 322	86
97	235	14

b. Escriba un programa en C++ que lea los datos en el archivo creado en el ejercicio 8a y despliegue el número de automóvil, las millas recorridas, los galones utilizados y las millas por galón para cada automóvil. La salida deberá contener el total de millas recorridas, el total de galones utilizados y el promedio de millas por galón para todos los automóviles. Estos totales deberán desplegarse al final del reporte de salida.

9. a. Cree un archivo de texto con los siguientes datos (sin los encabezados):

Número de parte	Cantidad inicial	Cantidad vendida	Cantidad mínima
QA310	95	47	50
CM145	320	162	200
MS514	34	20	25
EN212	163	150	160

b. Escriba un programa en C++ para crear un reporte de inventario basado en los datos en el archivo creado en el ejercicio 9a. El despliegue deberá consistir en el número de parte, balance actual y la cantidad que es necesaria para mantener el inventario en el nivel mínimo.

10. a. Cree un archivo de texto que contenga los siguientes datos (sin encabezados):

Nombre	Tarifa	Horas
Callaway,G.	6.00	40
Hanson,P.	5.00	48
Lasard,D.	6.50	35
Stillman,W.	8.00	50

b. Escriba un programa en C++ que use la información contenida en el archivo creado en el ejercicio 10a para producir el siguiente reporte de pagos para cada empleado:

```
Nombre  Tarifa tarifa  Horas  Pago bruto  Pago de tiempo extra  Pago regular
```

El pago regular se calculará como cualesquiera horas trabajadas hasta 40 horas inclusive, multiplicadas por la tarifa de pago. El pago de tiempo extra se calculará como cualesquiera horas trabajadas que rebasen las 40 horas por una tarifa de pa-

go de 1.5 multiplicada por la tarifa regular, y el pago bruto es la suma del pago regular y el pago de tiempo extra. Al final del reporte, el programa deberá desplegar los totales de las columnas de pago regular, de tiempo extra y bruto.

11. a. Almacene los siguientes datos en un archivo:

5 96 87 78 93 21 4 92 82 85 87 6 72 69 85 75 81 73

b. Escriba un programa en C++ para calcular y desplegar el promedio de cada grupo de números en el archivo creado en el ejercicio 11a. Los datos están ordenados en el archivo de modo que cada grupo de números esté precedido por el número de elementos de datos en el grupo. Por tanto, el primer número en el archivo, 5, indica que los siguientes cinco números deberán agruparse juntos. El número 4 indica que los siguientes cuatro números son un grupo, y el 6 indica que los últimos seis números son un grupo. (*Sugerencia:* Use un ciclo anidado. El ciclo exterior deberá terminar cuando se haya encontrado el final del archivo.)

8.3 EXCEPCIONES Y COMPROBACIÓN DE ARCHIVOS[1]

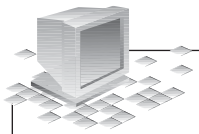
La detección de errores y su procesamiento con el manejo de excepciones se usa en forma extensa dentro de los programas en C++ que usan uno o más archivos. Por ejemplo, si un usuario elimina o cambia el nombre a un archivo usando un comando del sistema operativo, esta acción causará que un programa en C++ falle cuando una llamada a la función `open()` intente abrir el archivo bajo su nombre original.

Se recordará de la sección 7.1 que el código para el manejo de excepciones general luce como éste:

```
try
{
  // una o más instrucciones,
  // al menos una de las cuales deberá
  // lanzar una excepción
}
catch(tipo-de-datos-de-la-excepción nombre_parámetro)
{
  // una o más instrucciones
}
```

En este código, se ejecutan las instrucciones del `bloque try`. Si no ocurren errores, las instrucciones del bloque `catch` se omiten y el procesamiento continúa con la instrucción que sigue al bloque `catch`. Sin embargo, si cualquier instrucción dentro del bloque `try` lanza una excepción, se ejecuta el bloque `catch` cuyo tipo de datos de la excepción corresponda con ésta. Si no se definió un bloque `catch` para un bloque `try`, ocurre un error de compilador. Si no existe un bloque `catch` que atrape un tipo de datos lanzado, ocurre una caída del programa si se lanza la excepción. La mayor parte de las veces, el bloque `catch` despliega un mensaje de error y termina el procesamiento con una llamada a la función `exit()`. El programa 8.7 ilustra las instrucciones requeridas para abrir un archivo en modo de lectura que incluyan manejo de excepciones.

[1]Esta sección puede omitirse en la primera lectura sin perder la continuidad temática.



Programa 8.7

```
#include <iostream>
#include <fstream>
#include <cstdlib>   // necesario para exit()
#include <string>
using namespace std;

int main()
{
  string nombre_archivo = "precios.dat"; // pone al frente el nombre del archivo
  string descrip;
  double precio;

  ifstream archivo_entr;

  try  // este bloque trata de abrir el archivo, leerlo y desplegar
         los datos del archivo
  {
    archivo_entr.open(nombre_archivo.c_str());

    if (archivo_entr.fail()) throw nombre_archivo; // ésta es la excepción
    que se está comprobando

    // lee y despliega el contenido del archivo
    archivo_entr >> descrip >> precio;
    while (archivo_entr.good()) // comprueba el siguiente carácter
    {
      cout << descrip << ' ' << precio << endl;
      archivo_entr >> descrip >> precio;
    }
    archivo_entr.close();

    return 0;
  }
  catch (string e)
  {
    cout << "\nEl archivo " << e << " no se abrió con éxito"
         << "\n Por favor compruebe que el archivo existe en realidad."
         << endl;
     exit(1);
  }
}
```

Punto de información

Comprobar que un archivo se abrió con éxito

Usando manejo de excepciones, el método más común para comprobar que el sistema operativo localizó el archivo designado es el codificado en el programa 8.7, cuyos puntos clave de codificación se repiten aquí por comodidad:

```
try // este bloque trata de abrir el archivo, leerlo y desplegar los
       datos del archivo
{
  // abre el archivo, lanzando una excepción si la apertura falla
  // ejecuta todo el procesamiento del archivo requerido
  // cierra el archivo
}
catch (string e)
{
  cout << "\nEl archivo " << e << " no se abrió con éxito"
       << "\n Por favor compruebe que el archivo existe en realidad." << endl;
  exit(1);
}
```

El mensaje de excepción producido por el programa 8.7 cuando no se encontró el archivo `precios.dat` es el siguiente:

```
El archivo precios.dat no se abrió con éxito
 Por favor compruebe que el archivo existe en realidad.
```

Aunque el código de manejo de excepciones en el programa 8.7 puede usarse para comprobar una apertura de archivo exitosa para entrada y salida, por lo general se requiere una comprobación más rigurosa para los archivos de salida porque, en la salida, casi está garantizado que el archivo será encontrado. Si existe, el archivo se encontrará; si no existe, el sistema operativo lo creará (a menos que se haya especificado el modo anexar y el archivo exista, o el sistema operativo no pueda encontrar la carpeta indicada). Sin embargo, saber que el archivo se ha encontrado y abierto es insuficiente para propósitos de salida cuando un archivo de salida existente no debe sobrescribirse. En estos casos, el archivo puede abrirse para entrada y, es encontrado, puede hacerse una comprobación posterior para asegurar que el usuario proporciona de manera explícita la aprobación para sobrescribirlo. La forma en que se logra esto se ilustra en código resaltado dentro del programa 8.8.

Programa 8.8

```
#include <iostream>
#include <fstream>
#include <cstdlib> // necesario para exit()
#include <string>
#include <iomanip> // necesario para formatear
using namespace std;

int main()
{
  char respuesta;
  string nombre_archivo = "precios.dat"; // pone al frente el nombre del archivo
  ifstream archivo_entr;
  ofstream archivo_sal;

  try // abre un flujo de entrada básico simplemente para comprobar si
        el archivo existe
  {
    archivo_entr.open(nombre_archivo.c_str());
    if (archivo_entr.fail()) throw 1; // esto significa que el archivo
    no existe
      // sólo llega aquí si se encontró el archivo;
      // de lo contrario el bloque catch toma el control
    cout << "Un archivo con el nombre " << nombre_archivo << " existe
    actualmente.\n"
         << "Desea sobrescribirlo con los datos nuevos (si o no): ";
    cin >> respuesta;
    if (tolower(respuesta) == 'no')
    {
      archivo_entr.close();
      cout << "El archivo existente no se ha sobrescrito." << endl;
      exit(1);
    }
   }
   catch(int e) {};   // un bloque para no hacer nada que permita
                      // que continúe el procesamiento
   try
   {
     // abre el archivo en modo de escritura y continúa con la escritura
        del archivo
     archivo_sal.open(nombre_archivo.c_str());
     if (archivo_sal.fail()) throw nombre_archivo;
     // establece los formatos del flujo de archivo de salida
```

(Continúa)

(Continuación)

```
      archivo_sal << setiosflags(ios::fixed)
                  << setiosflags(ios::showpoint)
                  << setprecision(2);
      // escribe los datos en el archivo
      archivo_sal << "Alfombras " << 39.95 << endl
                  << "Bombillas " << 3.22 << endl
                  << "Fusibles " << 1.08 << endl;
      archivo_sal.close();
      cout << "El archivo " << nombre_archivo
           << " se ha escrito con éxito." << endl;


      return 0;
    }
    catch(string e)
    {
      cout << "El archivo " << nombre_archivo
           << " no se abrió para salida y no se ha escrito."
           << endl;
    }
}
```

En el programa 8.8, los `bloques try` están separados. Debido a que un bloque catch está afiliado con el `bloque try` previo más cercano, no hay ambigüedad respecto a bloques `try` y `catch` sin par.

Apertura de múltiples archivos

Como un ejemplo de la aplicación del manejo de excepciones en la apertura de dos archivos al mismo tiempo, supóngase que se desea leer los datos de un archivo basado en caracteres llamado `info.txt`, un carácter a la vez, y escribir estos datos en un archivo llamado `info.bak`. En esencia, esta aplicación es un programa de copia de archivos que lee los datos de un archivo carácter por carácter y escribe los datos en un segundo archivo. Como ilustración, suponga que los caracteres almacenados en el archivo de entrada son como se muestra en la figura 8.3.

```
Ahora es el momento en que todas las buenas personas
  acudan en ayuda de su partido.
Por favor llame al (555) 888-6666 para
  mayor informacion.
```

Figura 8.3 *Los datos almacenados en el archivo* `info.txt`.

La figura 8.4 ilustra la estructura de los flujos necesarios para producir la copia del archivo. En esta figura, un objeto de flujo de entrada referenciado por la variable `archi-`

vo_entr leerá datos del archivo `info.txt`, y un objeto de flujo de salida referenciado por la variable `archivo_sal` escribirá datos en el archivo `info.bak`.

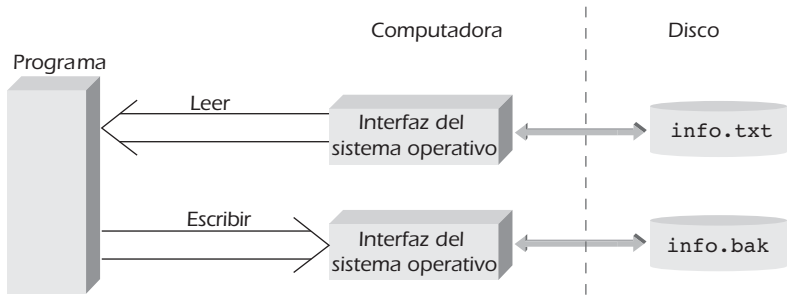


Figura 8.4 *Estructura del flujo de la copia del archivo.*

Ahora considere el programa 8.9, el cual crea el archivo `info.bak` como un duplicado exacto del archivo `info.txt` utilizando el procedimiento ilustrado en la figura 8.4.

Programa 8.9

```
#include <iostream>
#include <fstream>
#include <cstdlib>   // necesario para exit()
#include <string>
using namespace std;

int main()
{
  string archivoUno = "info.txt"; // pone al frente el nombre del archivo
  string archivoDos = "info.bak";
  char ch;
  ifstream archivo_entr;
  ofstream archivo_sal;

  try   // este bloque trata de abrir el archivo de entrada
  {
    // abre un flujo de entrada básico
    archivo_entr.open(archivoUno.c_str());
      if (archivo_entr.fail()) throw archivoUno;
  } // fin del bloque try exterior
  catch (string entrar) // catch para el bloque try exterior
```

(Continúa)

(Continuación)

```
  {
    cout << "El archivo de entrada " << entrar
         << " no se abrió con éxito." << endl
         << " No se hizo el respaldo." << endl;
    exit(1);
  }

  try // este bloque trata de abrir el archivo de salida y
  {    // ejecuta todo el procesamiento del archivo

    archivo_sal.open(archivoDos.c_str());
    if (archivo_sal.fail()) throw archivoDos;
    while (ch = archivo_entr.get())!= EOF)
       archivo_sal.put(ch);

    archivo_entr.close();
    archivo_sal.close();
   }
   catch (string salir)   // catch para el bloque try interior
   {
     cout << "El archivo de respaldo " << salir
          << " no se abrió con éxito." << endl;
     exit(1);
   }

  cout << "Un respaldo exitoso de " << archivoUno
       << " llamado " << archivoDos << " se realizó correctamente." << endl;

  return 0;
}
```

Por simplicidad, el programa 8.9 intenta abrir los archivos de entrada y salida dentro de bloques `try` separados y no anidados. De manera más general, el segundo archivo será abierto en un bloque `try` interior anidado de modo que el intento de abrir este segundo archivo no se haría si la apertura del primer archivo lanzara una excepción. (El Punto de información sobre anidación de bloques `try` explica cómo se logra esto.)

Punto de información

Anidación de bloques `try`

Cuando está implicado más de un flujo de archivo, abrir cada uno de ellos en su propio bloque `try` permite el aislamiento e identificación exactos de cuál archivo causó una excepción, si ocurre una. Los bloques `try` pueden anidarse. Por ejemplo, considere el programa 8.9, el cual se volvió a escribir aquí usando bloques `try` anidados. Hay que observar que en este caso el bloque `catch` para el bloque `try` interior debe anidarse en el mismo alcance de bloque que su bloque

```
#include <iostream>
#include <fstream>
#include <cstdlib>  // necesario para exit()
#include <string>
using namespace std;

int main()
{
  string archivoUno = "info.txt"; // pone al frente el nombre
                      del archivo
  string archivoDos = "info.bak";

  char ch;
  ifstream archivo_entr;
  ifstream archivo_sal;

  try // este bloque trata de abrir el archivo de entrada
  {
    // abre un flujo de entrada básico
    archivo_entr.open(archivoUno.c_str());
    if (archivo_entr.fail()) throw archivoUno;
    try  // este bloque trata de abrir el archivo de salida y
    {    // ejecuta todo el procesamiento del archivo

         // abre un flujo de salida básico
      archivo_sal.open(archivoDos.c_str());
      if (archivo_sal.fail()) throw archivoDos;
      while ((ch = archivo_entr.get()) != EOF)
        archivo_sal.put(ch);

      archivo_entr.close();
      archivo_sal.close();
    }  // fin del bloque try interior
    catch (string salir) // catch para el bloque try interior
```

(Continúa)

(Continuación)

```
    {
      cout << "El archivo de respaldo " << salir
           << " no se abrió con éxito." << endl
      exit(1);
    }
  }  // fin del bloque try exterior
  catch (string entrar) // catch para el bloque try exterior
  {
    cout << "El archivo de entrada " << entrar
         << " no se abrió con éxito." << endl;
         << "No se hizo ningun respaldo." << endl;
    exit(1);
  }

  cout << "Un respaldo exitoso de " << archivoUno
       << " llamado " << archivoDos << " se realizó correctamente."
          << endl;

  return 0;
}
```

El punto importante que hay que señalar en este programa es el anidamiento de los bloques `try`. Si los dos bloques `try` no estuvieran anidados y la declaración del flujo de entrada, `ifstream archivo_entr;` , se colocara en el primer bloque, no podría utilizarse en el segundo bloque `try` sin producir un error de compilador. La razón para esto es que todas las variables declaradas en un bloque de código, el cual es definido por un par de llaves de apertura y de cierre, son locales para el bloque en el que fueron declaradas.

Al revisar el programa 8.9, ponga particular atención a la instrucción:

```
while((ch = archivo_entr.get())!= EOF)
```

Esta instrucción lee en forma continua un valor del flujo de entrada hasta que se detecta el valor EOF. En tanto el valor devuelto no sea igual al valor EOF, el valor es escrito en el flujo del objeto de salida. Los paréntesis que rodean a la expresión `(ch = archivo_entr.get())` son necesarios para asegurar que el valor es leído primero y asignado a la variable `ch` antes que el valor recuperado se compare con el valor EOF. En su ausencia, la expresión completa sería `ch = archivo_entr.get()!= EOF`. Debido a la precedencia de las operaciones, la expresión relacional `archivo_entr.get()!= EOF` se ejecutaría primero. Debido a que ésta es una expresión relacional, su resultado es un valor booleano verdadero o falso basado en los datos recuperados por el método `get()`. Asignar este resultado booleano a la variable de carácter `ch` es una conversión inválida a través de un operador de asignación.

Ejercicios 8.3

1. Enliste dos condiciones que causen una condición de falla cuando un archivo es abierto para entrada.
2. Enliste dos condiciones que causen una condición de falla cuando un archivo es abierto para salida.
3. Si un archivo que existe es abierto para salida en modo de escritura, ¿qué les sucederá a los datos que están en el archivo?
4. Modifique el programa 8.7 para usar un identificador de su elección, en lugar de la letra `e`, para el nombre del parámetro de excepción del bloque `catch`.
5. Introduzca y ejecute el programa 8.8.
6. Determine por qué los dos bloques `try` en el programa 8.8, los cuales no están anidados, no causan problemas en la compilación o la ejecución. (*Sugerencia:* Coloque la declaración para el nombre del archivo dentro del primer bloque `try` y compile el programa.)
7. **a.** Si los bloques `try` anidados en el Punto de información sobre bloques `try` anidados se separan en bloques no anidados, el programa no se compilará. Determine por qué sucede esto.
   b. ¿Qué cambios adicionales tendría que hacer al programa en el ejercicio 7a que permitirían que se escribiera con bloques no anidados? (*Sugerencia:* Vea el ejercicio 6.)
8. Introduzca los datos para el archivo `info.txt` en la figura 8.3 u obténgalos en el sitio web de este texto (véase el Prefacio para la URL). Luego introduzca y ejecute el programa 8.9 y verifique que haya sido escrito el archivo de respaldo.
9. Modifique el programa 8.9 para usar un método `getline()` en lugar del método `get()` que se encuentra actualmente en el programa.

8.4 ARCHIVOS DE ACCESO ALEATORIO

El término **acceso a archivos** se refiere al proceso de recuperar datos de un archivo. Existen dos tipos de acceso a los archivos: acceso secuencial y acceso aleatorio. Para entender los tipos de acceso a los archivos, es necesario entender algunos conceptos relacionados con la forma en que los datos están organizados dentro de un archivo.

El término **organización de archivos** se refiere a la forma en que los datos están almacenados en un archivo. Los archivos que se han usado, y continuarán usándose, tienen una **organización secuencial.** Esto significa que los caracteres dentro del archivo están almacenados de una manera secuencial.

Además de estar organizados en forma secuencial, se ha leído cada archivo abierto de una manera secuencial. Es decir, se ha tenido acceso a cada carácter en secuencia. Esto se conoce como **acceso secuencial.** Sin embargo, aunque los caracteres en el archivo estén almacenados en forma secuencial, esto no nos obliga a tener acceso secuencial a ellos. De hecho, podemos saltar caracteres y leer un archivo organizado en forma secuencial de una manera no secuencial.

En el **acceso aleatorio,** cualquier carácter en el archivo abierto puede leerse en forma directa sin tener que leer primero en forma secuencial todos los caracteres almacenados antes que él. Para proporcionar acceso aleatorio a los archivos, cada objeto `ifstream` crea en forma automática un marcador de posición de archivo. Este marcador es un número entero largo que representa un desplazamiento desde el principio de cada archivo y le sigue la pista al lugar desde donde se va a leer o a escribir el siguiente carácter. Las funciones usadas para tener acceso y cambiar el marcador de posición del archivo se muestran en la tabla 8.4. Los sufijos `g` y `p` en estos nombres de función denotan `get` y `put`, respectivamente, donde `get` se refiere a un archivo de entrada (obtener desde) y `put` se refiere a un archivo de salida (poner en).

Tabla 8.4 Funciones de marcadores de posición del archivo

Nombre	Descripción
`seekg(offset, mode)`	Para archivos de entrada, se mueve a la posición de desplazamiento indicada por el modo.
`seekp(offset, mode)`	Para archivos de salida, se mueve a la posición de desplazamiento indicada por el modo.
`tellg(void)`	Para archivos de entrada, devuelve el valor actual del marcador de posición del archivo.
`tellp(void)`	Para archivos de salida, devuelve el valor actual del marcador de posición del archivo.

Las funciones `seek()` permiten al programador moverse a cualquier posición en el archivo. Para entender este método, debe entenderse cómo están referenciados los datos en el archivo usando el marcador de posición del archivo.

Cada carácter en un archivo de datos se localiza por su posición en el archivo. El primer carácter en el archivo se localiza en la posición 0, el siguiente en la posición 1, etc. Se hace referencia a la posición de un carácter como su desplazamiento desde el inicio del archivo. Por tanto, el primer carácter tiene un desplazamiento de 0, el segundo carácter tiene un desplazamiento de 1, etc., para cada carácter en el archivo.

Las funciones `seek()` requieren dos argumentos: el primero es el desplazamiento, como un número entero largo, en el archivo; el segundo es desde dónde se va a calcular el desplazamiento, lo que es determinado por el modo. Las tres alternativas posibles para el modo son `ios::beg`, `ios::cur` e `ios::end`, los cuales denotan el principio, la posición actual y el final del archivo, respectivamente. Por tanto, un modo de `ios::beg` significa que el desplazamiento es el desplazamiento verdadero desde el inicio del archivo. Un modo de `ios::cur` significa que el desplazamiento es relativo a la posición actual en el archivo, y un modo `ios::end` significa que el desplazamiento es relativo al final del archivo. Un desplazamiento positivo significa avanzar en el archivo y un desplazamiento negativo significa retroceder. A continuación se muestran ejemplos de llamadas a la función `seek()`. En estos ejemplos, suponga que se ha abierto `archivo_entr` como un archivo de entrada y `archivo_sal` como un archivo de salida. En estos ejemplos, el desplazamiento transmitido a `seekg()` y `seekp()` debe ser un número entero largo.

```
iarchivo_entr.seekg(4L, ios::beg); // va al quinto carácter en el archivo de entrada
archivo_sal.seekp(4L, ios::beg);   // va al quinto carácter en el archivo de salida
archivo_entr.seekg(4L, ios::cur);  // avanza cinco caracteres en el archivo de entrada
```

```
archivo_sal.seekp(4L, ios::cur);   // avanza cinco caracteres en el archivo de salida
archivo_entr.seekg(-4L, ios::cur);  // retrocede cinco caracteres en el archivo de entrada
archivo_sal.seekp(-4L, ios::cur);  // retrocede cinco caracteres en el archivo de salida
archivo_entr.seekg(0L, ios::beg);  // va al inicio del archivo de entrada
archivo_sal.seekp(0L, ios::beg);   // va al inicio del archivo de salida
archivo_entr.seekg(0L, ios::end);  // va al final del archivo de entrada
archivo_sal.seekp(0L, ios::end);   // va al final del archivo de salida
archivo_entr.seekg(-10L, ios::end);  //  va a 10 caracteres antes del final del archivo de
                                        entrada
archivo_sal.seekp(-10L, ios::end);   //  va a 10 caracteres antes del final del archivo de
                                        salida
```

A diferencia de las funciones `seek()` que mueven el marcador de posición del archivo, las funciones `tell()` devuelven el valor de desplazamiento del marcador de posición del archivo. Por ejemplo, si se han leído diez caracteres desde un archivo de entrada llamado `archivo_entr`, la llamada a la función devuelve el número entero largo 10:

```
archivo_entr.tellg();
```

Esto significa que el siguiente carácter que se va a leer está desplazado diez posiciones de byte del inicio del archivo y es el undécimo carácter en el archivo.

El programa 8.10 ilustra el uso de `seekg()` y `tellg()` para leer un archivo en orden inverso, del último carácter al primero. Conforme se lee cada carácter, también se despliega.

Programa 8.10

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
using namespace std;

int main()
{
  string nombre_archivo = "prueba.dat";
  char ch;
  long offset, last;

  ifstream archivo_entr(nombre_archivo.c_str());

  if (archivo_entr.fail())   // comprueba la apertura con exito
  {
    cout << "\nEl archivo no se abrió con éxito."
         << "\n Por favor compruebe que el archivo existe en realidad."
         << endl;
    exit(1);
  }
```

(Continúa)

(Continuación)

```
  archivo_entr.seekg(0L, ios::end);  // se mueve al final del archivo
  last = archivo_entr.tellg(); // guarda el desplazamiento del último carácter

  for(offset = 1L; offset <= last; offset++)
  {
    archivo_entr.seekg(-offset, ios::end);
    ch = archivo_entr.get();
    cout << ch << " : ";
  }

  archivo_entr.close();

  cout << endl;

  return 0;
}
```

Suponga que el archivo `prueba.dat` contiene los siguientes datos:

```
El grado fue 92.5
```

La salida del programa 8.10 es la siguiente:

```
5 : . : 2 : 9 :   : e : u : f :   : o : d : a : r : g :   : I : E :
```

El programa 8.10 va inicialmente al último carácter en el archivo. El desplazamiento de este carácter, el carácter EOF, es guardado en la variable `last`. Dado que `tellg()` devuelve un número entero largo, `last` ha sido declarado como un número entero largo.

Empezando desde el final del archivo, se usa `seekg()` para ubicar el siguiente carácter que se va a leer, referenciado desde el final del archivo. Conforme se lee cada carácter, el carácter es desplegado y el desplazamiento ajustado para tener acceso al siguiente carácter. El primer desplazamiento usado es −1, el cual representa al carácter que precede inmediatamente al marcador EOF.

Ejercicios 8.4

1. **a.** Cree un archivo llamado `prueba.dat` que contenga los datos que hay en el archivo `prueba.dat` usado en el programa 8.10. Puede hacer esto usando un editor de texto.
   b. Introduzca y ejecute el programa 8.10 en su computadora.
2. Vuelva a escribir el programa 8.10 de modo que el origen para la función `seekg()` usada en el ciclo `for` sea el inicio del archivo en lugar del final.
3. Modifique el programa 8.10 para desplegar un mensaje de error si `seekg()` intenta referenciar una posición más allá del final del archivo.

4. Escriba un programa que lea y despliegue cada segundo carácter en un archivo llamado `prueba.dat`.

5. Usando las funciones `seek()` y `tell()`, escriba una función llamada `carac_archivo()` que devuelva el número total de caracteres en un archivo.

6. **a.** Escriba una función llamada `leerBytes()` que lea y despliegue *n* caracteres empezando desde cualquier posición en un archivo. La función deberá aceptar tres argumentos: un nombre de objeto de archivo, el desplazamiento del primer carácter que se va a leer y el número de caracteres que se leerán. (*Nota*: El prototipo para `leerBytes()` deberá ser `void leerBytes(fstream&, long, int)`.)
   b. Modifique la función `leerBytes()` escrita en el ejercicio 6a para almacenar los caracteres leídos en una cadena o arreglo. La función deberá aceptar la dirección del área de almacenamiento como un cuarto argumento.

8.5 FLUJOS DE ARCHIVO COMO ARGUMENTOS DE FUNCIONES

Un objeto de flujo de archivo puede utilizarse como un argumento de función. El único requisito es que el parámetro formal de la función sea una referencia (véase la sección 6.3) al flujo apropiado, ya sea `ifstream&` u `ofstream&`. Por ejemplo, en el programa 8.11, un objeto `ofstream` llamado `archivo_sal` se abre en `main()` y este objeto de flujo es transmitido a la función in`Out()`. El prototipo de la función y la línea de encabezado para `inOut()` declaran el parámetro formal como una referencia a un tipo de objeto `ofstream`. La función in`Out()` se usa entonces para escribir en el archivo cinco líneas de texto introducido por el usuario.

Programa 8.11

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
using namespace std;


int main()
{
  string nombre_archivo = "lista.dat"; // aquí está el archivo con el que
                          estamos trabajando

  void inOut(ofstream&);    // prototipo de la función

  ofstream archivo_sal;
```

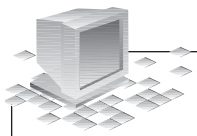
(Continúa)

(Continuación)

```
  archivo_sal.open(nombre_archivo.c_str());
  if (archivo_sal.fail())   // comprueba una apertura exitosa
  {
    cout << "\nEl archivo de salida " << nombre_archivo << " no se abrio
            con exito"

         << endl;
    exit(1);
  }

  inOut(archivo_sal);   // llama a la función

  return 0;
}

void inOut(ofstream& salArchivo)
{

  const int NUMLINEAS = 5; // número de líneas de texto
  string linea;
  int cuenta;

  cout << "Por favor introduzca cinco líneas de texto:" << endl;
  for (cuenta = 0; cuenta < NUMLINEAS; cuenta++)
  {
    getline(cin,linea);
    salArchivo << linea << endl;
  }

  cout << "\nEl archivo se ha escrito con éxito." << endl;
  return;
}
```

Dentro de `main()`, el archivo es un objeto `ostream` llamado `archivo_sal`. Este objeto es transmitido a la función `inOut()` y es aceptado como el parámetro formal llamado `salArchivo`, el cual es declarado como una referencia a un tipo de objeto `ostream`. La función `inOut()` usa entonces su parámetro de referencia `archivo_sal` como un nombre de flujo de archivo de salida de la misma manera que `main()` usaría el objeto de flujo `salArchivo`. El programa 8.11 usa el método `getline()` introducido en la sección 8.2 (véase la tabla 8.3).

En el programa 8.12 se ha expandido el programa 8.11 agregando una función `getOpen()` para ejecutar la apertura. `getOpen()`, como `inOut()`, acepta un argumento de referencia para un objeto `ofstream`. Después que la función `getOpen()` completa la ejecución, esta referencia es transmitida a `inOut()`, como en el programa 8.11. Aunque podría estar tentado a escribir `getOpen()` para que devuelva una referencia a un `ofstream`, esto no funcionará porque produce un intento de asignar una referencia devuelta a una ya existente.



Programa 8.12

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
using namespace std;

int main()
{
  int getOpen(ofstream&);  // transmite una referencia a un fstream
  void inOut(ofstream&);   // transmite una referencia a un fstream

  ofstream archivo_sal;   // el nombre de archivo es un objeto fstream

  getOpen(archivo_sal);   // abre el archivo
  inOut(archivo_sal);     // escribe en el

  return 0;
}

int getOpen(ofstream& salArchivo)
{
  string nombre;

  cout << "\nIntroduzca un nombre de archivo: ";
  getline(cin,nombre);

  salArchivo.open(nombre.c_str());   // abre el archivo

  if (salArchivo.fail())   // comprueba una apertura exitosa
  {
    cout << "No se pudo abrir el archivo" << endl;
    exit(1);
  }
  else
    return 1;
}
```

(Continúa)

(Continuación)

```
void inOut(ofstream& salArchivo)
{
  const int NUMLINEAS = 5; // número de líneas
  int cuenta;
  string linea;

  cout << "Por favor introduzca cinco líneas de texto:" << endl;
  for (cuenta = 0; cuenta < NUMLINEAS; cuenta++)
  {
    getline(cin,linea);
    salArchivo << linea << endl;
  }
  cout << "\nEl archivo se ha escrito con éxito." << endl;
  return;
}
```

El programa 8.12 es una versión modificada del programa 8.11 que permite al usuario introducir un nombre de archivo desde el dispositivo de entrada estándar y luego abre la conexión `ofstream` al archivo externo. Si se introduce el nombre de un archivo de datos existente, el archivo será destruido cuando se abra para salida. Un truco útil que puede emplearse para prevenir este tipo de percance es abrir el archivo introducido usando un flujo de archivo de entrada. Si el archivo existe, el método `fail()` indicará una apertura exitosa (es decir, la apertura no fallará), lo cual indica que el archivo está disponible para entrada. Esto puede usarse para alertar al usuario que existe en el sistema un archivo con el nombre introducido y para solicitar confirmación de que los datos en el archivo pueden destruirse y el archivo abrirse para salida. Antes que el archivo sea reabierto para salida, deberá cerrarse el flujo de archivo de entrada. La puesta en práctica de este algoritmo se deja como ejercicio.

Ejercicios 8.5

1. Una función llamada `p_archivo()` recibirá un nombre de archivo como una referencia a un objeto `ifstream`. ¿Qué declaraciones se requieren para transmitir un nombre de archivo a `p_archivo()`?

2. Escriba una función, llamada `revisar_archivo()`, que compruebe si existe un archivo. La función aceptará un objeto `ifstream` como un parámetro de referencia formal. Si el archivo existe, deberá devolver un valor de 1; de lo contrario, la función deberá devolver un valor de cero.

3. Suponga que se ha creado un archivo de datos consistente en un grupo de líneas individuales. Escriba una función llamada `impr_linea()` que leerá y desplegará cualquier línea deseada del archivo. Por ejemplo, la llamada a la función `impr_linea(fstream& nombre_archivo,5)`; deberá desplegar la quinta línea del flujo de objeto transmitido.

4. Vuelva a escribir la función `getOpen()` usada en el programa 8.12 para incorporar los procedimientos de comprobación de archivo descritos en esta sección. De manera específica, si existe el nombre de archivo introducido, deberá desplegarse un mensaje apropiado. Al usuario deberá presentársele la opción de introducir un

nombre de archivo nuevo o permitir que el programa sobrescriba el archivo existente. Use la función escrita para el ejercicio 2 en su programa.

8.6 ERRORES COMUNES DE PROGRAMACIÓN

Los errores comunes de programación con respecto a los archivos son:

1. Usar un nombre externo de archivo en lugar del nombre de objeto de flujo de archivo interno cuando se tiene acceso al archivo. El único método de flujo que usa el nombre externo del archivo de datos es la función `open()`. Como siempre, todos los métodos de flujo presentados en este capítulo deben ser precedidos por un nombre de objeto de flujo y el operador punto.
2. Abrir un archivo para salida sin comprobar primero que ya existe un archivo con el nombre dado. No comprobar un nombre de archivo preexistente asegura que el archivo será sobrescrito.
3. No entender que el fin de archivo sólo se detecta después que se ha leído o transmitido el centinela EOF.
4. Intentar detectar el final de un archivo usando variables de carácter para el marcador EOF. Cualquier variable usada para aceptar el EOF debe ser declarada como una variable en entero. Por ejemplo, si se declara `ch` como una variable de carácter, la siguiente expresión produce un ciclo infinito.[2]

   ```
   while ( (ch = in.archivo.peek()) != EOF )
   ```

   Esto ocurre debido a que una variable de carácter nunca puede tomar un código EOF. El EOF es un valor entero (por lo general −1) que no tiene representación de carácter. Esto asegura que el código EOF nunca pueda confundirse con ningún carácter legítimo encontrado como dato normal en el archivo. Para terminar el ciclo creado por la expresión anterior, la variable `ch` debe declararse como una variable en número entero.
5. Usar un argumento en número entero con las funciones `seekg()` y `seekp()`. Este desplazamiento debe ser una constante o variable en número entero largo. Cualquier otro valor transmitido a estas funciones puede producir un efecto imprevisible.

8.7 RESUMEN DEL CAPÍTULO

1. Un archivo de datos es cualquier colección de datos almacenados juntos en un medio de almacenamiento externo bajo un nombre común.
2. Un archivo de datos se conecta a un flujo de archivos usando el método `open()` de `fstream`. Esta función conecta un nombre externo de archivo con un nombre de objeto interno. Después que se ha abierto el archivo, todos los accesos subsiguientes al archivo requieren el nombre del objeto interno.

[2]Esto no ocurrirá en el sistema UNIX, ya que los caracteres se almacenan como enteros con signo.

3. Un archivo puede abrirse en modo de entrada o de salida. Un flujo de archivo de salida abierto crea un archivo de datos nuevo o elimina los datos en un archivo abierto existente. Un flujo de archivo de entrada abierto hace que los datos en un archivo existente estén disponibles para entrada. Se produce una condición de error si el archivo no existe y puede detectarse usando el método `fail()`.
4. Todos los flujos de archivo deben declararse como objetos de las clases `ifstream` u `ofstream`. Esto significa que debe incluirse una declaración similar a cualquiera de las siguientes con las que se abre el archivo:

   ```
   ifstream archivo_entr;
   ofstream archivo_sal;
   ```

   Los nombres de objeto de flujo `archivo_entr` y `archivo_sal` pueden reemplazarse con cualquier nombre de objeto seleccionado por el usuario.
5. Además de los archivos abiertos dentro de una función, los objetos de flujo estándar `cin`, `cout` y `cerr` son declarados y abiertos en forma automática cuando se ejecuta un programa. `cin` es el nombre de objeto de un flujo de archivo de entrada usado para la introducción de datos (por lo general desde el teclado), `cout` es el nombre de objeto de un flujo de archivo de salida usado para desplegar datos por omisión (por lo general en la pantalla de la computadora) y `cerr` es el nombre de objeto de un flujo de archivo de salida usado para desplegar mensajes de error del sistema (por lo general en la pantalla de la computadora).
6. Puede tenerse acceso aleatorio a los archivos de datos usando los métodos `seekg()`, `seekp()`, `tellg()` y `tellp()`. Las versiones `g` de estas funciones se usan para alterar y buscar el marcador de posición del archivo para flujos de archivo de entrada, y las versiones `p` hacen lo mismo para los flujos de archivo de salida.
7. La tabla 8.5 muestra los métodos suministrados por la clase `fstream` para la manipulación de archivos.

Tabla 8.5 Métodos `fstream`

Nombre del método	Descripción
`get()`	Extrae el siguiente carácter del flujo de entrada y lo devuelve como un `int`.
`get(chrVar)`	Extrae el siguiente carácter del flujo de entrada y lo asigna a `chrVar`.
`getline(fileObj, string, termChar)`	Extrae la siguiente cadena de caracteres del objeto de flujo de entrada y la asigna a la cadena hasta que se detecte el carácter de terminación especificado. Si se omite, el carácter de terminación por omisión es una línea nueva.
`getline(C-stringVar,int n,'\n')`	Extrae y devuelve caracteres del flujo de entrada hasta que se han leído `n-1` caracteres o se encuentra una línea nueva (termina la entrada con un `'\0'`).
`peek()`	Devuelve el siguiente carácter en el flujo de entrada sin extraerlo del flujo.
`put(chrExp)`	Pone el carácter especificado por `chrExp` en el flujo de salida.
`putback(chrExp)`	Pone de vuelta el carácter especificado por `chrExp` en el flujo de entrada. No altera los datos en el archivo.

Tabla 8.5 Métodos `fstream` (continuación)

Nombre del método	Descripción
`ignore(int n)`	Se salta los siguientes *n* caracteres; si se omite `n`, el valor por omisión es saltarse el siguiente carácter individual.
`eof()`	Devuelve un valor booleano `verdadero` si se ha intentado una lectura después del final del archivo; de lo contrario, devuelve un valor booleano `falso`. El valor se vuelve verdadero sólo cuando se lee el primer carácter después del último carácter de archivo válido.
`good()`	Devuelve un valor booleano `verdadero` mientras el archivo está disponible para uso del programa. Devuelve un valor booleano `falso` si se ha intentado una lectura después del final del archivo. El valor se vuelve falso sólo cuando se lee el primer carácter después del último carácter de archivo válido.
`bad()`	Devuelve un valor booleano `verdadero` si se ha intentado una lectura después del final del archivo; de lo contrario, devuelve un `falso`. El valor se vuelve verdadero sólo cuando se lee el primer carácter después del último carácter de archivo válido.
`fail()`	Devuelve un booleano `verdadero` si el archivo no se ha abierto con éxito; de lo contrario, devuelve un valor booleano `falso`.

8.8 COMPLEMENTO DEL CAPÍTULO: LA BIBLIOTECA DE CLASE `iostream`

Como se ha visto, las clases contenidas dentro de la biblioteca de clase `iostream` tienen acceso a los archivos usando entidades llamadas flujos. Para la mayor parte de los sistemas, los bytes de datos transferidos a un flujo representan caracteres ASCII o números binarios.

El mecanismo para leer un flujo de byte de un archivo o escribir un flujo de byte a un archivo está oculto cuando se usa un lenguaje de nivel superior como C++. No obstante, es útil entender este mecanismo de modo que se puedan colocar los servicios proporcionados por la biblioteca de la clase iostream en su contexto apropiado.

Mecanismo de transferencia de flujo de archivos

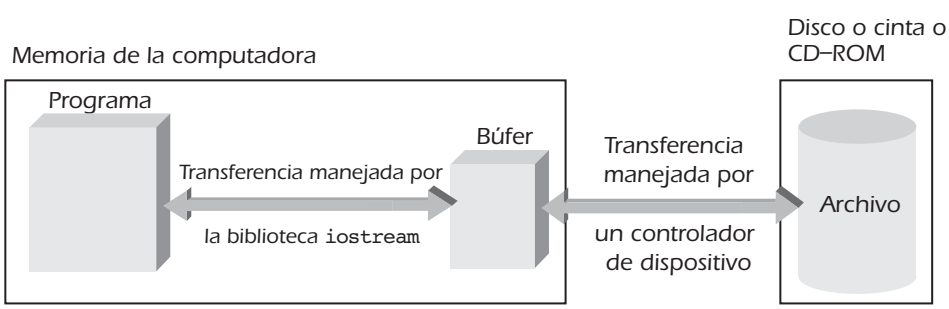


Figura 8.5 *El mecanismo de transferencia de datos.*

El mecanismo para transferir datos entre un programa y un archivo se ilustra en la figura 8.5. Como se puede observar, transferir datos entre un programa y un archivo implica un búfer de archivo intermediario contenido en la memoria de la computadora. A cada archivo abierto se le asigna su propio búfer de archivo, el cual es un área de almacenamiento utilizado por los datos que se están transfiriendo entre el programa y el archivo.

Desde este lado, el programa escribe un conjunto de bytes de datos al búfer del archivo o lee un conjunto de bytes de datos del búfer del archivo usando un objeto de flujo. En el otro lado del búfer, la transferencia de datos entre el dispositivo que almacena el archivo de datos real (por lo general una cinta, disco o CD-ROM) y el búfer del archivo es manejado por programas especiales del sistema operativo a los que se hace referencia como controladores de dispositivo. Los controladores de dispositivo no son programas autónomos sino parte integral del sistema operativo. Un **controlador de dispositivo** es una sección de código del sistema operativo que tiene acceso a un dispositivo de hardware, como una unidad de disco, y maneja la transferencia de datos entre el dispositivo y la memoria de la computadora. Como tal, debe sincronizar en forma correcta la velocidad de los datos transferidos entre la computadora y el dispositivo que envía o recibe los datos. Esto se debe a que la velocidad de transferencia interna de la computadora por lo general es mucho más rápida que cualquier dispositivo conectado a ella.

Por lo general, un controlador de dispositivo de disco sólo transferirá datos entre el disco y el búfer de archivo en tamaños fijos, como 1024 bytes a la vez. Por tanto, el búfer del archivo proporciona un medio conveniente para permitir a un controlador de dispositivo transferir datos en bloques de un tamaño, y el programa puede tener acceso a ellos usando un tamaño diferente (generalmente, como caracteres individuales o como un número fijo de caracteres por línea).

Componentes de la biblioteca de clase `iostream`

La biblioteca de clase `iostream` consiste en dos clases primarias base: la clase `streambuf` y la clase `ios`. La clase `streambuf` proporciona el búfer de archivo, ilustrado en la figura 8.5, y diversas rutinas generales para transferir datos binarios. La clase `ios` contiene un apuntador a los búfer de archivo proporcionados por la clase `streambuf` y diversas rutinas generales para transferir datos de texto. A partir de estas dos clases base, se derivan otras clases diversas y se incluyen en la biblioteca de clase `iostream`.

La figura 8.6 ilustra un diagrama de herencia para la familia de clases `ios` y su relación con las clases `ifstream, ofstream` y `fstream`. El diagrama de herencia para la familia de clases `streambuf` se muestra en la figura 8.7. La convención adoptada para los diagramas de herencia es que las flechas apuntan de una clase derivada a una clase base.

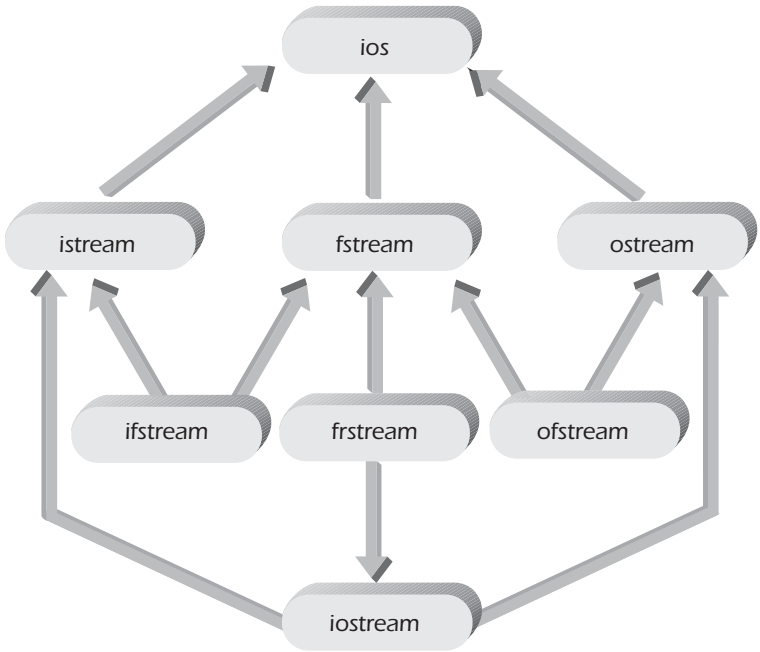


Figura 8.6 *La clase base* `ios` *y sus clases derivadas.*

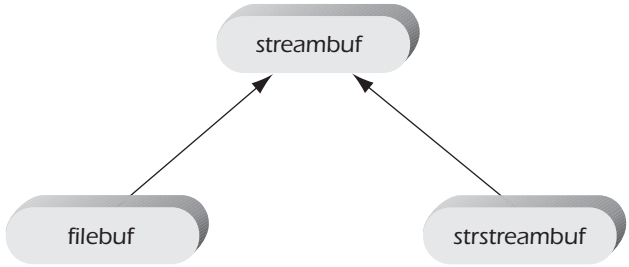


Figure 8.7 *La clase base* `streambuf` *y sus clases derivadas.*

La correspondencia entre las clases ilustradas en las figuras 8.6 y 8.7, incluyendo los archivos de encabezado que definen estas clases, se muestran en la tabla 8.6.

Tabla 8.6 Correspondencia entre las clases ilustradas en las figuras 8.6 y 8.7

Clase ios	Clase streambuf	Archivos de encabezado
istream ostream iostream	streambuf	iostream o fstream
ifstream ofstream fstream	filebuf	fstream

Por tanto, las clases `ifstream, ofstream` y `fstream` que se han usado para el acceso a archivos usan un búfer proporcionado por la clase `filebuf`, definida en el archivo de encabezado `fstream`. Del mismo modo, los objetos `iostream` `cin`, `cout`, `cerr` y `clog` que se han usado a lo largo del texto usan un búfer proporcionado por la clase `streambuf` y definido en el archivo de encabezado `iostream`.

Formateo en memoria

Además de las clases ilustradas en la figura 8.6, también se deriva de la clase `ios` una clase llamada `strstream`. Esta clase usa la clase `strstreambuf` ilustrada en la figura 8.7, requiere el archivo de encabezado `strstream` y proporciona capacidades para escribir y leer cadenas desde y hacia flujos definidos en memoria.

Como flujo de salida, tales flujos se utilizan por lo general para "ensamblar" una cadena de piezas más pequeñas hasta que una línea completa de caracteres está lista para ser escrita, ya sea a `cout` o a un archivo. Anexar un objeto `strstream` a un búfer para este propósito se hace de manera similar a la anexión de un objeto `fstream` a un archivo de salida. Por ejemplo, la instrucción

```
strstream inmem(buf, 72, ios::out);
```

anexa un objeto `strstream` a un búfer existente de 72 bytes en modo de salida. El programa 8.13 ilustra cómo se usa esta instrucción dentro del contexto de un programa completo.

Programa 8.13

```
#include <iostream>
#include <strstream>
#include <iomanip>
using namespace std;

int main()
{
  const int MAXCARAC = 81; // uno más que el máximo de caracteres en una línea
  int unidades = 10;
  double precio = 36.85;
  char buf[MAXCARAC];

  strstream inmem(buf, MAXCARAC, ios::out); // abre un flujo en la memoria

  // escribe al bufer a través del flujo
  inmem << "Num. de unidades = "
        << setw(3) << units
        << "  Precio por unidad = $ "
        << setw(6) << setprecision(2) << fixed << precio << '\0';

  cout << '|' << buf << '|';

  cout << endl;

  return 0;
}
```

La salida producida por el programa 8.13 es la siguiente:

```
|Num. de unidades = 10 Precio por unidad = $ 36.85|
```

Esta salida ilustra que el búfer de caracteres se ha llenado en forma correcta por las inserciones al flujo `inmem`. (Hay que observar que el final de la cadena `NULL, '\0'`, el cual es la última inserción en el flujo, se requiere para cerrar en forma correcta la cadena C.) Una vez que se ha llenado el arreglo de caracteres deseado, se escribirá en un archivo como una sola cadena.

De manera similar, un objeto `strstream` puede abrirse en modo de entrada. Dicho flujo se usaría como un área de trabajo de almacenamiento, o búfer, para almacenar una línea completa de texto de un archivo o entrada estándar. Una vez que se ha llenado el búfer, se usará el operador de extracción para "desensamblar" la cadena en sus partes componentes y convertir cada elemento de datos en su tipo de datos designado. Hacer esto permite introducir datos de un archivo línea por línea antes de asignar los elementos de datos individuales a sus variables respectivas.

CAPÍTULO 9
Introducción a las clases

TEMAS

9.1 TIPOS DE DATOS ABSTRACTOS EN C++ (CLASES)
- TIPOS DE DATOS ABSTRACTOS
- CONSTRUCCIÓN DE CLASES
- TERMINOLOGÍA

9.2 CONSTRUCTORES
- CONSTRUCTORES QUE LLAMAN
- CONSTRUCTORES SOBRECARGADOS Y EN LÍNEA
- DESTRUCTORES

9.3 APLICACIONES
- APLICACIÓN 1: CONSTRUIR UN OBJETO DE HABITACIÓN
- APLICACIÓN 2: SIMULACIÓN DE UNA BOMBA DE GASOLINA CON UN OBJETO SIMPLE

9.4 ERRORES COMUNES DE PROGRAMACIÓN

9.5 RESUMEN DEL CAPÍTULO

Además de ser una versión mejorada de C, la característica distintiva de C++ es su apoyo a la programación orientada a objetos. Es fundamental para esta orientación a objetos el concepto de un tipo de datos abstracto, el cual es definido por el programador. En este capítulo se abordarán las implicaciones de permitir a los programadores definir sus propios tipos de datos y luego se presentará el mecanismo de C++ para construirlos. Como se verá, la construcción de un tipo de datos se basa tanto en variables como en funciones; las variables proporcionan el medio para crear configuraciones de datos nuevas y las funciones proporcionan el medio para ejecutar operaciones en estas estructuras. Lo que proporciona C++ es una forma única de combinar variables y funciones en una unidad independiente y cohesiva a partir de la cual puedan crearse objetos.

Punto de información

Lenguajes de procedimientos, híbridos y puros orientados a objetos

La mayor parte de los lenguajes de programación de nivel alto pueden clasificarse en una de tres categorías principales: de procedimientos, híbridos, u orientados a objetos. FORTRAN, que fue el primer lenguaje de programación de nivel alto disponible en forma comercial, es de procedimientos. Esto tiene sentido porque FORTRAN fue diseñado para ejecutar cálculos matemáticos que usan fórmulas algebraicas estándar. Estas fórmulas se describieron como algoritmos y luego los algoritmos se codificaron usando procedimientos de función y subrutinas. Otros lenguajes de procedimientos que siguieron a FORTRAN incluyeron BASIC, COBOL y Pascal.

En la actualidad sólo hay dos lenguajes puros orientados a objetos: Smalltalk y Eiffel. El primer requisito de un lenguaje puro orientado a objetos es que contiene tres características específicas: clases, herencia y polimorfismo (cada una de estas características se describe en éste y en el siguiente capítulo). Sin embargo, además de proporcionar estas características, un lenguaje "puro" orientado a objetos, como mínimo, siempre debe usar clases. En un lenguaje puro orientado a objetos todos los tipos de datos se construyen como clases, todos los valores de datos son objetos, todos los operadores pueden ser sobrecargados y cada operación con datos sólo puede ejecutarse usando una función que es miembro de una clase. En un lenguaje puro orientado a objetos es imposible no usar características orientadas a objetos a través de un programa. Esto no sucede en un lenguaje híbrido.

En un lenguaje híbrido, como C++, *es imposible no usar elementos de un programa de procedimientos*. Esto se debe a que el uso de cualquier tipo de datos u operación integrados viola en efecto el paradigma puro orientado a objetos. Aunque un lenguaje híbrido debe tener la capacidad de definir clases, la característica distintiva de un lenguaje híbrido es que es posible escribir un programa completo usando sólo código de procedimientos. Además, los lenguajes híbridos ni siquiera necesitan proporcionar herencia y características polimórficas, pero deben proporcionar clases. Los lenguajes que usan clases pero no proporcionan herencia y características polimórficas se conocen como lenguajes *basados en objetos* en lugar de *orientados a objetos*.

9.1 TIPOS DE DATOS ABSTRACTOS EN C++ (CLASES)

Vivimos en un mundo lleno de objetos: aviones, trenes, automóviles, teléfonos, libros, computadoras, etc. Sin embargo, hasta hace bastante poco, las técnicas de programación no habían reflejado esto. El paradigma principal de programación[1] había sido de procedimientos, definiendo un programa como un algoritmo escrito en un lenguaje que fuera legible para la máquina. Las razones para este énfasis en la programación de procedimientos son sobre todo históricas.

Cuando las computadoras fueron desarrolladas en la década de 1940 eran usadas por los matemáticos con propósitos militares, para calcular la trayectoria de las bombas y descifrar las órdenes y transmisiones diplomáticas del enemigo. Después de la Segunda Guerra Mundial, aún eran utilizadas sobre todo por matemáticos para hacer sus cálculos. Esta realidad se reflejó en el nombre del primer lenguaje de nivel alto disponible en forma comercial introducido en 1957. El nombre del lenguaje era FORTRAN, el cual era un acrónimo para FORmula TRANslation (traducción de fórmulas). Un reflejo más de este uso

[1] Un *paradigma* es una forma estándar de pensar en algo o de hacer algo.

predominante fue el hecho que en la década de 1960 casi todos los cursos de computación se impartían en los departamentos de ingeniería o matemáticas. El término ciencia de la computación todavía no era de uso común y los departamentos de ciencia de la computación apenas estaban en formación

Esta situación ha cambiado de manera considerable, sobre todo por dos razones. Una de las razones del desencanto por los programas orientados a procedimientos ha sido el fracaso de los lenguajes tradicionales de procedimientos para proporcionar un medio adecuado para limitar los costos del software. Los costos del software incluyen todos los costos asociados al desarrollo inicial del programa y el mantenimiento subsiguiente del mismo. Como se ilustra en la figura 9.1, el costo principal de la mayor parte de los proyectos de cómputo en la actualidad, sean técnicos o comerciales, es el software.

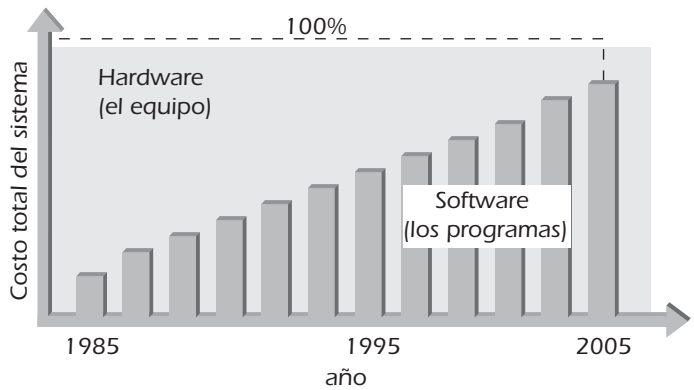


Figura 9.1 *El software es el costo principal de la mayor parte de los proyectos de cómputo.*

Los costos del software contribuyen de manera tan importante a los costos totales del proyecto debido a que se relacionan en forma directa con la productividad humana (son de trabajo intensivo), mientras el equipo asociado con los costos del hardware se relaciona con las tecnologías de manufactura. Por ejemplo, los microchips que costaban más de 500 dólares hace diez años ahora pueden comprarse por menos de un dólar.

Sin embargo, es mucho más fácil incrementar de manera considerable la productividad de manufactura en un 1000%, con la consiguiente disminución en los costos del hardware, que lograr que los programadores dupliquen la cantidad o la calidad del código que producen. Así que aunque los costos del hardware se han desplomado, la productividad del software y sus costos asociados han permanecido relativamente constantes. Por tanto, la proporción de los costos del software respecto a los costos totales del sistema (hardware más software) ha aumentado de manera considerable.

Una forma de aumentar de manera significativa la productividad del programador es crear código que pueda reutilizarse sin una revisión, prueba y revalidación extensas. La incapacidad del código estructurado por procedimientos para proporcionar este tipo de reutilización ha conducido a la búsqueda de otros enfoques de software.

La segunda razón del desencanto por la programación basada en procedimientos ha sido el surgimiento de pantallas gráficas y el subsiguiente interés en las aplicaciones con ventanas. Proporcionar una interfaz gráfica de usuario (GUI) en la cual un usuario pueda moverse con facilidad incluso en una sola ventana es un desafío utilizando código de procedimientos. Programar ventanas múltiples y quizá superpuestas en la misma pantalla gráfica aumenta enormemente la complejidad cuando se usa código de procedimientos.

A diferencia del enfoque por procedimientos, sin embargo, un enfoque orientado a objetos se adapta bien a los ambientes con ventanas gráficas, donde cada ventana puede ser espesificada como un objeto rectangular independiente que puede moverse y redimensionarse en relación con otros objetos en la pantalla. Además, dentro de cada ventana pueden colocarse y moverse con facilidad otros objetos gráficos, como cuadros de verificación, botones de opción, etiquetas y cuadros de texto.

Para proporcionar esta capacidad de creación de objetos, se elaboraron extensiones al lenguaje de procedimientos C. Esta extensión se convirtió en el nuevo lenguaje llamado C++, el cual permite a un programador usar y crear objetos nuevos.

Para la creación de objetos el concepto de un tipo de datos abstracto es central, el cual tan sólo es un tipo de datos definido por el usuario, en oposición a los tipos de datos integrados proporcionados por todos los lenguajes (como los tipos en número entero y en punto flotante). El hecho de permitir al programador definir nuevos tipos de datos, como por ejemplo un tipo rectangular, del cual pueden crearse y desplegarse en la pantalla objetos rectangulares específicos, forma la base de la orientación a objetos de C++.

Tipos de datos abstractos

Para obtener una comprensión clara de lo que es un tipo de datos abstracto, considere los siguientes cuatro tipos de datos integrados incluidos en C++: números enteros, números de precisión doble, booleanos y caracteres. Al usar estos tipos de datos por lo general se declaran una o más variables del tipo deseado, utilizadas en sus formas aceptadas y evitando usarlas en formas que no están especificadas. Por tanto, por ejemplo, no se usaría el operador de módulo en dos números de precisión doble. Debido a que esta operación no tiene sentido para números de precisión doble, nunca se define, en ningún lenguaje de programación, para dichos números. Por tanto, aunque por lo general no se considera, cada tipo de datos consiste tanto de un *tipo* de datos, un número entero o uno de precisión doble, *como* de capacidades operacionales específicas proporcionadas para cada tipo.

En terminología de computación, la combinación de datos y sus operaciones asociadas se define como un **tipo de datos**. Es decir, un tipo de datos define tanto los tipos de datos como los tipos de operaciones que pueden ejecutarse con ellos. Visto bajo esta luz, el tipo de datos de números enteros, el tipo de datos de números de precisión doble y el tipo de datos de caracteres proporcionados en C++ son todos ejemplos de tipos de datos **integrados** que se definen por el tipo de datos y capacidades operacionales específicas proporcionadas para inicializar y manipular el tipo. En forma simplificada esta relación puede describirse como

Tipo de datos = valores de datos permisibles + capacidades operacionales

Por tanto, las operaciones que hemos visto utilizadas en C++ son una parte inherente de cada tipo de datos que se han utilizado. Para cada uno de estos tipos de datos los diseñadores de C++ tuvieron que considerar con minuciosidad, y luego poner en práctica, operaciones específicas.

Para entender la importancia de las capacidades operacionales proporcionadas por un lenguaje de programación, tomemos un momento para enlistar algunas de las suministradas con los tipos de datos integrados de C++ (`int`, `double`, `bool` y `char`). El conjunto mínimo de las capacidades proporcionadas por los tipos de datos integrados de C++ muestran en la tabla 9.1.[2]

[2]Quizá oueda observar la ausencia de operaciones de lectura y escritura. Tanto en C como en C++, excepto para operaciones muy primitivas, la entrada y salida es proporcionada por rutinas de biblioteca y funciones de clase estándar.

Tabla 9.1 *Capacidades de los tipos de datos integrados de C++*

Capacidad	Ejemplo
Define una o más variables del tipo de datos	`int a, b;`
Inicializa una variable en definición	`int a = 5;`
Asigna un valor a una variable	`a = 10;`
Asigna un valor de una variable a otra variable	`a = b;`
Ejecuta operaciones matemáticas	`a + b`
Ejecuta operaciones relacionales	`a > b`
Convierte de un tipo de datos a otro	`a = int (7.2);`

Ahora veamos cómo se relaciona todo esto con los tipos de datos abstractos (TDA). Por definición un **tipo de datos abstracto** es simplemente un tipo definido por el usuario que define tanto un tipo de datos como las operaciones que pueden ejecutarse en ellos. Estos tipos de datos definidos por el usuario se requieren cuando se desea crear objetos que son más complejos que números enteros y caracteres simples. Si vamos a crear nuestros propios tipos de datos debemos estar conscientes tanto del tipo de datos que estamos creando como de las capacidades que proporcionaremos para inicializarlos y manipularlos.

Como un ejemplo específico, suponga que se está programando una aplicación que utiliza fechas en forma extensa. Es claro que, desde un punto de vista de datos, una fecha debe ser capaz de tener acceso y almacenar una designación de mes, día y año. Aunque desde un punto de vista de la implementación hay diversos medios de almacenar una fecha, desde el punto de vista del usuario la implementación real no es relevante. Por ejemplo, una fecha puede almacenarse como tres números enteros, uno para el mes, día y año, respectivamente. De manera alternativa, también puede usarse un solo número entero largo en la forma *aaaammdd*. Usando la implementación del número entero largo la fecha 5/16/08 se almacenaría como el número entero 20080516. Para ordenar fechas, el formato de número entero largo es muy atractivo debido a que la secuencia numérica de las fechas corresponde a la secuencia del calendario.

El método de estructurar la fecha en forma interna, por desgracia, sólo suministra una respuesta parcial al esfuerzo de programación. Aún se debe suministrar un conjunto de operaciones que puedan usarse con fechas. Es evidente que tales operaciones podrían asignar valores a una fecha, restar dos fechas para determinar el número de días entre ellas, comparar dos fechas para determinar cuál es anterior y cuál es posterior, y desplegar una fecha en una forma como 12/03/06 en lugar de 12/3/6.

Hay que observar que los detalles de cómo funciona cada operación dependen de la manera que se elija para almacenar una fecha (lo que se conoce de manera formal como su estructura de datos) y es sólo de interés para nosotros cuando desarrollamos cada operación. Por ejemplo, la implementación de comparar dos fechas diferirá si se almacena una fecha usando un número entero largo en oposición a usar números enteros separados para el mes, día y año, respectivamente.

La combinación de la estructura de almacenamiento utilizada para las fechas con un conjunto de operaciones disponibles apropiadas para fechas definiría entonces un tipo de datos Fecha abstracto. Una vez que se elabora este tipo de fecha, los programadores que deseen usarlo no tienen que preocuparse nunca por cómo se almacenan las fechas o cómo se ejecutan las operaciones. Todo lo que necesitan saber es qué hace cada operación y cómo

invocarla, en forma parecida a como se usan las operaciones integradas en C++. Por ejemplo, en realidad no nos importa cómo se ejecuta la adición de dos números enteros sino sólo que esto se hace en forma correcta.

En C++ un tipo de datos abstracto se conoce como una **clase**. La construcción de una clase es inherentemente fácil y ya se tienen todas las herramientas necesarias en variables y funciones. Lo que proporciona C++ es un mecanismo para empacar juntos estos dos elementos en una unidad independiente. Veamos cómo se hace esto.

Construcción de clases

Una clase define tanto datos como funciones. Esto se logra por lo general al construir una clase en dos partes, consistentes en una sección de declaración y una sección de implementación. Como se ilustra en la figura 9.2, la sección de declaración declara tanto los tipos de datos como las funciones para la clase. Luego se usa la sección de implementación para definir las funciones cuyos prototipos se han declarado en la sección de declaración.[3] Cuando una función es parte de una clase, se conoce de manera formal como un método para denotar la pertenencia a la clase. Por convención, sin embargo, los términos método de clase y función de clase se usan de manera intercambiable en C++.

```
// sección de declaración de clase
class nombre_clase
{
  miembros de los datos // las variables
  miembro de la función // prototipos
};
// seccion de implementación de clase
definiciones de función
```

Figura 9.2 Formato de una definición `class`.

Tanto las variables como las funciones enlistadas en la sección de declaración de clase se conocen de manera colectiva como ***miembros de la clase***. De manera individual, las variables se conocen como ***miembros de datos*** y ***variables de instancia*** (los términos son sinónimos), mientras las funciones se conocen como ***funciones miembro***. El nombre de una función miembro no puede ser igual al nombre de un miembro de datos.

Como un ejemplo específico de una clase, considérese la siguiente definición de una clase llamada `Fecha`. Este tipo de clase es muy importante en aplicaciones donde las fechas de entrega de equipo y los pagos de intereses dependen de la determinación de las fechas. Usando dicha clase, la determinación de si una fecha cae en fin de semana o en día festivo, por ejemplo, se responde con facilidad.

[3] Esta separación en dos partes no es obligatoria ya que la implementación puede incluirse dentro de la sección de declaración, como se describe en la siguiente sección.

```
//--- sección de declaración de clase

class Fecha
{
  private:         // observe los dos puntos después de la palabra private
     int mes;     // un miembro de datos
     int dia;     // un miembro de datos
     int anio;    // un miembro de datos
   public:         // de nuevo, observe los dos puntos aquí
     Fecha(int = 7, int = 4, int = 2005);    // una función miembro, constructor
     void estFecha(int, int, int);     // una función miembro
     void mostrarFecha();             // una función miembro
};

//--- seccion de implementación de la clase

Fecha::Fecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;
}

void Fecha::estFecha(int mm, int dd, int aaaa)
{
  mes = mm; dia = dd; anio = aaaa;

  return;
}

void Fecha::mostrarFecha()
{
  cout << "La fecha es ";
  cout << setfill('0')
       << setw(2) << mes << '/'
       << setw(2) << dia << '/'
       << setw(2) << anio % 100; // extrae los últimos dos dígitos del año
  cout << endl;

  return;
}
```

Debido a que esta definición puede parecer abrumadora al principio, hay que observar primero que consta de dos secciones: una sección de declaración y una sección de implementación. Ahora considérese cada una de estas secciones de manera individual.

La sección de declaración de clase comienza con la palabra clave `class` seguida por un nombre de clase. Después del nombre de la clase están las declaraciones de variables y

prototipos de funciones de la clase, encerrados entre un par de llaves que terminan con un punto y coma. Por tanto, la estructura general de la forma que se ha usado es[4]

```
class Nombre
{
  private:
    una lista de declaraciones de variables
  public:
    una lista de prototipos de funcion
};
```

Hay que observar que este formato es seguido por nuestra clase `Fecha`, la cual por comodidad se ha enlistado a continuación sin comentarios internos.

```
//--- sección de declaración de clase

class Fecha
{
  private:
     int mes;
     int dia;
     int anio;
 public:
     Fecha(int = 7, int = 4, int = 2005);
     void estFecha(int, int, int);
     void mostrarFecha();
};  // esta es una declaracion, no olvide el punto y coma
```

El nombre de esta clase es `Fecha`. Aunque no se requiere la letra mayúscula inicial, se usa de manera convencional para designar una clase. El cuerpo de la sección de declaración, el cual se encierra dentro de llaves, consiste de declaraciones de variables y de función. En este caso los miembros de datos `mes`, `dia` y `anio` son declarados como números enteros y se declaran tres funciones llamadas `Fecha()`, `estFecha()` y `mostrarFecha()` por medio de prototipos. Las palabras clave `private` y `public` son especificadores de acceso que definen derechos de acceso. La palabra clave `private` especifica que sólo se puede tener acceso a los miembros de la clase que siguen, en este caso los miembros de datos `mes`, `dia` y `anio`, usando las funciones de clase (o funciones amigas, como se expondrá en la sección 9.2).[5] El propósito de la designación `private` pretende de manera específica imponer seguridad en los datos requiriendo que todo el acceso a los miembros de datos `private` sea a través de las funciones miembro proporcionadas. Este tipo de acceso, el cual restringe que un usuario vea la forma en que los datos están almacenados en realidad, se conoce como **ocultamiento de datos**. Una vez que se designa una categoría de clase como `private`, permanece en vigor hasta que se enlista una categoría nueva.

[4]Son posibles otras formas. Como esta forma es una de las más usadas y se entiende con facilidad, servirá como un modelo estándar a lo largo del texto.

[5]Hay que observar que la categoría de pertenencia por omisión en una clase es **`private`**, lo cual significa que esta palabra clave puede omitirse. En este texto se usará de manera explícita la designación **`private`** para reforzar la idea de restricciones de acceso inherentes en la pertenencia a una clase.

De manera específica, se ha elegido almacenar una fecha usando tres números enteros: uno para el mes, día y año, respectivamente. También se almacenará siempre el año como un número de cuatro dígitos. Por tanto, por ejemplo, se almacenará el año 1998 como 1998 y no como 98. Al asegurarse de almacenar todos los años con su designación del siglo correcta se eliminará una multitud de problemas que pueden surgir si sólo se almacenan los últimos dos dígitos, como 98. Por ejemplo, el número de años entre 2006 y 1999 puede calcularse rápido como 2006 – 1999 = 7 años, mientras que esta misma respuesta no se obtiene con facilidad si sólo se usan los valores de años 06 y 99. Además, estamos seguros a qué se refiere el año 2006, mientras que un valor de dos dígitos como 06 podría referirse a 1906 o a 2006.

Después de los miembros de datos de la clase `private`, los prototipos de función enlistados en la clase `Fecha` se han declarado como `public`. Esto significa que estas funciones de clase *pueden* ser llamadas por los objetos y funciones que no estén en la clase (externos). En general, todas las funciones de clase deberían ser `public`; como tales suministran capacidades para manipular las variables de clase desde fuera de la clase. Para la clase `Fecha` se han proporcionado inicialmente tres funciones llamadas `Fecha()`, `estFecha()` y `mostrarFecha()`. Hay que observar que una de estas funciones miembro tiene el mismo nombre, `Fecha`, que el nombre de la clase. Esta función particular se conoce como una función **constructora** y tiene un propósito definido en especial: puede usarse para inicializar miembros de datos de clase con valores. Los valores por omisión que se usan para esta función son los números 7, 4 y 2001, los cuales, como se verá pronto, se usan como los valores `mes`, `dia` y `anio` por omisión, respectivamente. El único punto a notar aquí es que el año por omisión está representado en forma correcta como un número entero de cuatro dígitos que conserva la designación del siglo. También hay que observar que la función constructora no tiene tipo de devolución, lo cual es un requisito para esta función especial. Las dos funciones restantes declaradas en el ejemplo de declaración son `estFecha()` y `mostrarFecha()`, las cuales han sido declaradas para no devolver ningún valor `(void)`. En la sección de implementación de la clase estas tres funciones miembro se escribirán para permitir capacidades de inicialización, asignación y despliegue, respectivamente.

La **sección de implementación** de una clase es donde se escriben las funciones miembro declaradas en la sección de declaración.[6] La figura 9.3 ilustra la forma general de las funciones incluidas en la sección de implementación. Este formato es correcto para todas las funciones excepto la constructora, la cual, como se ha expuesto, no tiene un tipo para devolver.

```
tipoDevuelto nombreClase::nombreFuncion(lista de parametros)
{
    cuerpo de la función
}
```

Figura 9.3 Formato de una función miembro.

[6]También es posible definir estas funciones dentro de la sección de declaración al declararlas y escribirlas como funciones en línea. En la sección 9.2 se presentan ejemplos de funciones miembro en línea.

Como se muestra en la figura 9.3, las funciones miembro definidas en la sección de implementación tienen el mismo formato que todas las funciones de C++ escritas por el usuario con la adición del nombre de clase y el operador de resolución de alcance, **::**, que identifica la función como un miembro de una clase particular. Ahora se reconsiderará la sección de implementación de la clase Fecha, la cual se repite a continuación por comodidad.

```
//--- sección de implementación de la clase

Fecha::Fecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;
}

void Fecha::estFecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;

  return;
}

void Fecha::mostrarFecha()
{
  cout << "La fecha es ";
  cout << setfill('0')
       << setw(2) << mes << '/'
       << setw(2) << dia << '/'
       << setw(2) << anio % 100; // extrae los últimos dos digitos
          del anio
  cout << endl;

  return;
}
```

Hay que observar que la primera función en esta sección de implementación tiene el mismo nombre que la clase, lo cual la hace una función constructora. Como tal, carece de un tipo para devolver. La `Fecha::` incluida en el principio de la línea de encabezado de la función identifica a esta función como un miembro de la clase Fecha. El resto de la línea de encabezado,

```
Fecha(int mm, int dd, int aaaa)
```

define que la función tiene tres parámetros de número entero. El cuerpo de esta función tan sólo asigna a los miembros de los datos `mes`, `dia` y `anio` los valores de los parámetros `mm`, `dd` y `aaaa`, respectivamente.

La siguiente línea de encabezado de la función

```
void Fecha::estFecha(int mm, int dd, int aaaa)
```

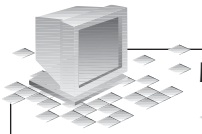
define a ésta como la función `estFecha()` perteneciente a la clase Fecha `(Fecha::)`. Esta función no devuelve ningún valor `(void)` y espera tres parámetros en número entero, `mm, dd` y `aaaa`. De una manera similar a la función `Fecha()`, el cuerpo de esta función les asigna a los miembros de datos mes, día y año los valores de sus parámetros.

Por último, la línea de encabezado final de la función en la sección de implementación define una función llamada `mostrarFecha()`. Esta función no tiene parámetros, no devuelve ningún valor y es un miembro de la clase `Fecha`. El cuerpo de esta función, sin embargo, necesita un poco más de explicación.

Aunque se ha elegido en forma interna almacenar todos los años como valores de cuatro dígitos que conserven información del siglo, los usuarios están acostumbrados a ver fechas donde el año es representado como un valor de dos dígitos, como 12/15/99. Para desplegar los últimos dos dígitos del valor del año, puede usarse la expresión 1999 % 100. Por ejemplo si el año es 1999, la expresión 1999 % 100 produce el valor 99, y si el año es 2006, la expresión 2006 % 100 produce el valor 6. Hay que observar que si se ha usado una asignación como `anio = anio % 100`; en realidad se alteraría el valor almacenado de `anio` para que correspondiera a los últimos dos dígitos del año. Dado que se desea conservar el año como un número de cuatro dígitos, debe tenerse cuidado en sólo manipular el valor desplegado usando la expresión `anio` % 100 dentro del flujo `cout`. Los manipuladores `setfill` y `setw` se utilizan para asegurar que los valores desplegados corresponden a fechas aceptadas de manera convencional. Por ejemplo, la fecha marzo 9, 2006, debería aparecer como 3/9/06 o 03/09/06. El manipulador `setw` obliga a que se despliegue cada valor en un ancho de campo de 2. Dado que este manipulador sólo tiene en efecto para la siguiente inserción, se ha incluido antes del despliegue de cada valor de datos. Sin embargo, como el manipulador `setfill` permanece en efecto hasta que se cambia el carácter de relleno, sólo se ha incluido una vez. Aqui se ha usado el manipulador `setfill` para cambiar el carácter de relleno de su valor por omisión de un espacio en blanco al carácter 0. Hacer esto asegura que una fecha como diciembre 9, 2006, aparecerá como `12/09/06` y no `12/ 9/ 6`.

Para ver cómo puede usarse la clase `Fecha` dentro del contexto de un programa completo, considérese el programa 9.1. Para faciltar la lectura del programa se ha sombreado en áreas más claras y más oscuras. El área más clara contiene las secciones de declaración de clase y de implementación que ya se han considerado. El área más oscura contiene el encabezado y la función `main()`. Por comodidad se conservará esta convención de sombreado para todos los programas que usen clases.[7]

[7]Este sombreado no es accidental. En la práctica la región con sombreado más claro que contiene la definición de clase se colocaría en un archivo separado. Entonces se utilizaría una sola instrucción `#include` para establecer esta declaración de clase en el programa. Por tanto, el programa final constaría de las dos regiones con sombreado más oscuro ilustradas en el programa 9.1 con la adición de una instrucción `#include` más en la primera región.



Programa 9.1

```
#include <iostream>
#include <iomanip>
using namespace std;

// declaración de clase

class Fecha
{
  private:
    int mes;
    int dia;
    int anio;
  public:
    Fecha(int = 7, int = 4, int = 2005); // constructor
    void estFecha(int, int, int); // funcion miembro para copiar una fecha
    void mostrarFecha();        // funcion miembro para desplegar una fecha
};

// seccion de implementación

Fecha::Fecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;
}
void Fecha::estFecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;

  return;
}
void Fecha::mostrarFecha()
{
   cout << "La fecha es ";
   cout << setfill('0')
        << setw(2) << mes << '/'
        << setw(2) << dia << '/'
        << setw(2) << anio % 100; // extrae los últimos dos dígitos del anio
   cout << endl;
```

(Continúa)

(Continuación)

```
  return;
}

int main()
{
  Fecha a, b, c(4,1,2000);  // declara tres objetos

  b.estFecha(12,25,2006); // asigna valores a los miembros de datos de b
  a.mostrarFecha();       // despliega los valores del objeto a
  b.mostrarFecha();       // despliega los valores del objeto b
  c.mostrarFecha();       // despliega los valores del objeto c

  return 0;
}
```

Las secciones de declaración e implementación contenidas en la región con sombreado más claro del programa 9.1 deberían serle familiares, ya que contienen las secciones de declaración e implementación de clase que ya se han expuesto. Hay que observar, sin embargo, que esta región sólo declara la clase; no crea ninguna variable de este tipo de clase. Esto sucede con todos los tipos de C++, incluyendo los tipos integrados como números enteros y de precisión doble. Del mismo modo en que una variable de un tipo entero debe definirse, también deben definirse las variables de una clase declarada por un usuario. Las variables definidas como parte de una clase declarada por un usuario se conocen como **objetos**.

Usando esta nueva terminología, la primera instrucción en la función `main()` del programa 9.1, contenida en el área más oscura, define que tres objetos, llamados `a`, `b` y `c`, son del tipo de la clase `Fecha`. En C++ siempre que se define un objeto nuevo se asigna memoria para éste y los miembros de datos se inicializan de manera automática. Esto se hace debido a una llamada automática a la función constructora `class`. Por ejemplo, considere la definición `Fecha a, b, c(4,1,2000);` contenida en `main()`. Cuando se define el objeto llamado `a`, la función constructora `Fecha` es invocada en forma automática. Debido a que no se han asignado parámetros a `a`, se usan los valores por omisión de la función constructora, produciendo la inicialización:

```
a.mes = 7
a.dia = 4
a.anio = 2005
```

Observe la notación que se ha usado aquí. Consiste en un nombre de objeto y un nombre de atributo separados por un punto. Ésta es la sintaxis estándar para referirse a un atributo de un objeto, a saber

`nombreObjeto.nombreAtributo`

donde *`nombreObjeto`* es el nombre de un objeto específico y *`nombreAtributo`* es el nombre de un miembro de datos definido por la clase del objeto.

Por tanto, la notación `a.mes = 7` se refiere al hecho que el miembro de datos mes del objeto `a` se ha establecido con un valor 7. Del mismo modo, la notación `a.dia = 4` y `a.anio = 2005` se refiere al hecho que los miembros de datos día y año de `a` se han establecido con valores `4` y `2005`, respectivamente. De la misma manera, cuando se defi-

ne el objeto llamado `b`, se usan los mismos parámetros por omisión, dando como resultado la inicialización de los miembros de datos de `b` como

```
b.mes = 7
b.dia = 4
b.anio = 2005
```

El objeto llamado `c`, sin embargo, se define con los argumentos 4, 1 y 2000. Estos tres argumentos son transmitidos a la función constructora cuando se define el objeto, lo que da como resultado la inicialización de los miembros de datos de `c` como

```
c.mes = 4
c.dia = 1
c.anio = 2000
```

La siguiente instrucción en `main()`, `b.estFecha(12,25,2006)`, llama a la función estFecha de `b`, la cual asigna los valores de argumento `12`, `25`, `2006` a los miembros de datos de `b`, lo que produce la asignación

```
b.mes = 12
b.dia = 25
b.anio = 2006
```

Hay que observar la sintaxis para referirse a un método de un objeto. Esta sintaxis es

```
nombreObjeto.nombreMétodo(parámetros)
```

donde *`nombreObjeto`* es el nombre de un objeto específico y *`nombreMétodo`* es el nombre de una de las funciones definidas para la clase del objeto. En vista que se han definido todas las funciones de clase como public, una instrucción como `b.estFecha(21,25,2006)` es válida dentro de la función `main()` y es una llamada a la función `estFecha()` de class. Esta instrucción le dice a la función `estFecha()` que opere en el objeto `b` con los argumentos 12, 25 y 2006. Es importante entender que debido a que todos los miembros de datos de la clase fueron especificados como `private`, una instrucción como `b.mes = 12` sería inválida desde dentro de `main()`. Por consiguiente, nos vemos obligados a depender de funciones miembro para tener acceso a los valores de los miembros de datos.

Las últimas tres instrucciones en `main()` llaman a la función `mostrarFecha()` para que opere en los objetos `a`, `b` y `c`. La primera llamada produce el despliegue de los valores de datos de `a`, la segunda llamada el despliegue de los valores de datos de `b` y la tercera llamada el despliegue de los valores de datos de `c`. Por tanto, la salida del programa 9.1 es

```
La fecha es 07/04/05
La fecha es 12/25/06
La fecha es 04/01/00
```

Hay que observar que una instrucción como `cout << a;` es inválida dentro de `main()` debido a que `cout` no sabe cómo manejar un objeto de la clase `Fecha`. Por tanto, se le ha suministrado a la clase una función que puede usarse para tener acceso a los valores internos de un objeto y desplegarlos.

Punto de información

Interfaces, implementaciones y ocultamiento de información

Los términos interfaz e implementación se usan de manera extensa en la literatura de programación orientada a objetos. Cada uno de estos términos puede equipararse a partes específicas de las secciones de declaración e implementación de una clase.

Una *interfaz* consiste de declaraciones de función miembro `public` de una clase y los argumentos de soporte. Como tal, la interfaz debería ser todo lo que se requiere para decirle a un programador cómo usar la `clase`.

La *implementación* consiste en la sección de implementación de la clase, la cual consiste de definiciones de miembros `private` y `public` y de los miembros de datos `private de la clase`, los cuales están contenidos en una sección de declaración de la clase.

La implementación es el medio esencial para *ocultar información*. En su contexto más general, el *ocultamiento de información* se refiere al principio de que la forma como está construida una clase en forma internamente no es relevante para el programador que desee usar la clase. Es decir, la implementación puede y deberá ocultarse de todos los usuarios de la clase precisamente para asegurar que ésta no es alterada o comprometida en alguna forma. Todo lo que necesita saber un programador para utilizar en forma correcta la clase deberá ser proporcionado por la interfaz.

Terminología

Puesto que en oaciones hay confusión acerca de los términos clases, objetos y otra terminología asociada con la programación orientada a objetos, tomaremos un momento para aclarar y revisar la terminología.

Una **clase** es un tipo de datos definido por un programador del cual pueden crearse objetos. Los **objetos** se crean de las clases; tienen la misma relación con las clases que las variables con los tipos de datos integrados de C++. Por ejemplo, en la declaración

```
int a;
```

se dice que `a` es una variable, mientras que en la declaración del programa 9.1

```
Fecha a;
```

se dice que a es un objeto. Si al principio le ayuda pensar en un objeto como una variable, hágalo así.

Los objetos también son conocidos como **instancias** de una clase y el proceso de crear un objeto nuevo con frecuencia se conoce como una **instanciación** del objeto. Cada vez que un objeto nuevo es instanciado (creado), se crea un conjunto nuevo de miembros de datos pertenecientes al objeto.[8] Los valores particulares contenidos en estos miembros de datos determinan el **estado** del objeto.

Visto de esta manera, una clase puede considerarse como una guía a partir de la cual pueden crearse instancias (objetos) particulares. Cada instancia (objeto) de una clase tendrá su propio conjunto de valores particulares para el conjunto de miembros de datos especificados en la sección de declaración de la clase.

Además de los tipos de datos permitidos para un objeto, una clase también define el

[8]Hay que observar que sólo se crea un conjunto de funciones de clase. Estas funciones son compartidas entre objetos.

comportamiento; es decir, las operaciones que se permite que se ejecuten en los miembros de datos de un objeto. Los usuarios del objeto necesitan saber *qué* pueden hacer estas funciones y cómo activarlas a través de llamadas a la función, pero a menos que el tiempo de ejecución o implicaciones de espacio sean relevantes, no necesitan saber cómo se hace la operación. Los detalles reales de la implementación de las operaciones de un objeto están contenidos en la implementación de la clase, la cual puede estar oculta al usuario. Otros nombres para las operaciones definidas en la sección de implementación de una clase son procedimientos, funciones, servicios y métodos. Estos términos se usarán de manera intercambiable a lo largo del resto del texto.

Ejercicios 9.1

1. Defina los siguientes términos:
   - **a.** clase
   - **b.** objeto
   - **c.** sección de declaración
   - **d.** sección de implementación
   - **e.** variable de instancia
   - **f.** función miembro
   - **g.** miembro de datos
   - **h.** constructor
   - **i.** instancia de clase
   - **j.** servicios
   - **k.** métodos
   - **l.** interfaz
   - **m.** estado
   - **n.** comportamiento

2. **a.** En lugar de especificar la ubicación de un rectángulo enlistando la posición de dos puntos de esquina diagonales, ¿qué otros atributos podría usar?
   b. ¿Qué otros atributos, además del largo y ancho, podrían usarse para describir un rectángulo si éste se va a dibujar en un monitor de color?
   c. Describa un conjunto de atributos que podrían usarse para definir círculos que van a dibujarse en un monitor en blanco y negro.
   d. ¿Qué atributos adicionales agregaría a aquellos seleccionados en respuesta al ejercicio 2c si los círculos fueran a dibujarse en un monitor a color?

3. **a.** Los atributos de una clase representan la forma en que los objetos de la clase se muestran ente el mundo exterior. El comportamiento representa cómo reacciona un objeto de una clase ante un estímulo externo. Dado esto, ¿cuál piensa que es el mecanismo por el cual un objeto "desencadena" el comportamiento designado en otro objeto?
   b. Si el comportamiento en C++ se construye al definir una función apropiada, ¿cómo piensa que se activa el comportamiento en C++?

4. Escriba una sección de declaración de clase para cada una de las siguientes especificaciones. En cada caso incluya un prototipo para un constructor y una función miembro llamada `mostrarDatos()` que pueda usarse para desplegar los valores de los miembros.
   a. Una clase llamada `Tiempo` que tenga miembros de datos en número entero llamados `segundos`, `minutos` y `horas`.
   b. Una clase llamada `Complejo` que tenga miembros de datos en número de precisión doble llamados `real` e `imaginario`.
   c. Una clase llamada `Circulo` que tenga miembros de datos en número entero llamados `centro_x` y `centro_y` y un miembro de datos en número de precisión doble llamado `radio`.
   d. Una clase llamada `Sistema` que tenga miembros de datos de carácter llama-

dos `computadora`, `impresora` y `pantalla`, cada uno capaz de contener 30 caracteres (incluyendo el final de la cadena `NULL`) y miembros de datos en número de precisión doble llamados `precioComp`, `precioImpr` y `precioPant`.

5. **a.** Construya una sección de implementación de clase para el constructor y los miembros de la función `mostrarDatos()` correspondientes a la declaración de clase creada para el ejercicio 4a.
   b. Construya una sección de implementación de clase para el constructor y los miembros de la función `mostrarDatos()` correspondientes a la declaración de clase creada para el ejercicio 4b.
   c. Construya una sección de implementación de clase para el constructor y los miembros de la función `mostrarDatos()` correspondientes a la declaración de clase creada para el ejercicio 4c.
   d. Construya una sección de implementación de clase para el constructor y los miembros de la función `mostrarDatos()` correspondientes a la declaración de clase creada para el ejercicio 4d.

6. **a.** Incluya las secciones de declaración e implementación de clase preparadas para los ejercicios 4a y 5a en un programa funcional completo.
   b. Incluya las secciones de declaración e implementación de clase preparadas para los ejercicios 4b y 5b en un programa funcional completo.
   c. Incluya las secciones de declaración e implementación de clase preparadas para los ejercicios 4c y 5c en un programa funcional completo.
   d. Incluya las secciones de declaración e implementación de clase preparadas para los ejercicios 4d y 5d en un programa funcional completo.

7. Determine los errores en la siguiente sección de declaración de clase:

```
class empleado
  {
  public:
    int num_emp;
    char codigo;
  private:
    class(int = 0);
    void mostrarEmp(int, char);
  };
```

8. **a.** Construya una clase llamada `Rectangulo` que tenga miembros de datos en número de precisión doble llamados `largo` y `ancho`. La clase deberá tener una función miembro llamada `perimetro()` y `area()` para calcular el perímetro y el área de un rectángulo, una función miembro llamada `estDatos()` para establecer el `largo` y `ancho` de los rectángulos y una función miembro llamada `mostrarDatos()` que despliegue el `largo`, `ancho`, `perimetro` y `area de un rectángulo`.
   b. Incluya la clase `Rectangulo` construida en el ejercicio 8a dentro de un programa en C++ funcional.

9. **a.** Modifique la clase `Fecha` definida en el programa 9.1 para incluir una función

`diaSig()` que incremente una fecha en un día. Pruebe su función para asegurarse que incrementa de manera correcta los días en un nuevo mes y en un nuevo año.

b. Modifique la clase `Fecha` definida en el programa 9.1 para incluir una función `diaAnt()` que disminuya una fecha en un día. Pruebe su función para asegurarse que disminuye en forma correcta los días en un mes anterior y en un año anterior.

10. Modifique la clase `Fecha` en el programa 9.1 para que contenga un método que compare dos objetos `Fecha` y devuelva el mayor de los dos. El método deberá escribirse de acuerdo con el siguiente algoritmo:

```
Función de comparación
    Acepta dos valores Fecha como parámetros
    Determina la fecha más reciente mediante el siguiente procedimiento:
        Convierte cada fecha en un valor entero que tiene la forma aaaammdd
        Esto puede lograrse usando la fórmula año*100000 + mes * 100 + día
        Compara los enteros correspondientes para cada fecha
        El número entero más grande corresponde a la fecha más reciente
    Devuelve la fecha más reciente
```

11. a. Agregue una función miembro a la definición de clase del programa 9.1 que determine el día de la semana para cualquier objeto fecha. Un algoritmo para determinar el día de la semana, conocido como algoritmo de Zeller, es el siguiente

Para fechas con la forma mm/dd/ssaa, donde mm es el mes, dd es el día, ss es el siglo y aa es el año en el siglo. Por ejemplo, con 12/28/2006, mm = 12, dd = 28, ss = 20 y aa = 6.

```
Si mm es menor que 3
    Establecer mm = mm + 12 y ssaa = ssaa-1
Endif
Establecer ss = int(ssaa/100)
Establecer aa = ssaa % 100
Establecer la variable T = dd + int(26 * (mm + 1)/10 + aa + int(aa/4) -2 * ss
diaSemana = T % 7
Si diaSemana es menor que 0
    diaSemana = diaSemana + 7
Endif
```

Usando este algoritmo, el `dia-de-la-semana` tendrá un valor de 0 si la fecha es un sábado, 1 si es un domingo, etcétera.

b. Incluya la definición de clase construida para el ejercicio 11a en un programa en C++ completo. La función `main()` deberá desplegar el nombre del día (Dom, Lun, Mar, etc.) para el objeto `Fecha` que se está probando.

9.2 CONSTRUCTORES

Una función **constructora** es cualquier función que tenga el mismo nombre que su clase. Pueden definirse múltiples constructores para cada clase en tanto sean distinguibles por el número y tipos de sus parámetros.

El propósito deseado de un constructor es inicializar los miembros de datos de un objeto nuevo. Como tal, dependiendo del número y tipos de argumentos suministrados, una función constructora es invocada de manera automática cada vez que se crea un objeto. Si no se escribe ninguna función constructora, el compilador suministra un constructor por omisión. Además de su papel de inicialización, una función constructora también puede ejecutar otras tareas cuando es llamada y puede escribirse en una variedad de formas. En esta sección se presentan las posibles variaciones de las funciones constructoras y se introduce otra función, la destructora, la cual es invocada de manera automática siempre que deja de existir un objeto.

La figura 9.4 ilustra el formato general de un constructor. Como se muestra en esta figura, un constructor

- debe tener el mismo nombre que la clase a la que pertenece
- no debe tener tipo a devolver (ni siquiera void)

```
nombreClase::nombreClase(lista de parámetros)
{
  cuerpo de la función
}
```

Figura 9.4 *Formato del constructor.*

Si no se incluye un constructor en la definición de clase, el compilador suministra uno por omisión que no hace nada. Por ejemplo, considere la siguiente declaración de clase:

```
class Fecha
{
  private:
    int mes, dia, anio;
  public:
    void estFecha(int, int, int);
    void mostrarFecha(void)
};
```

Debido a que no se ha declarado aquí un constructor definido por el usuario, el compilador crea un constructor por omisión. Para la clase `Fecha` este constructor por omisión es equivalente a la implementación `Fecha::Fecha(void){}`. Es decir, el constructor por omisión suministrado por el compilador no espera parámetros y tiene un cuerpo vacío. Es evidente que este constructor por omisión no es muy útil, pero existe si no se declara ningún otro constructor.

Punto de información

Constructores

Un **constructor** es cualquier función que tenga el mismo nombre que su clase. El propósito principal de un constructor es inicializar las variables de miembro de un objeto cuando se crea un objeto. Como tal, un constructor es invocado de manera automática cuando se declara un objeto.

Una `class` puede tener múltiples constructores bajo la condición que cada constructor sea distinguible por tener una lista de parámetros formales diferente. Cuando no es posible la identificación única de un constructor se produce un error de compilador. Si no se proporciona un constructor el compilador suministrará un constructor por omisión que no hace nada.

Cada función constructora debe declararse *sin tipo a devolver* (ni siquiera void). Puesto que son funciones, los constructores también pueden ser llamados en forma explícita en instrucciones no declarativas. Cuando se usan de esta manera, la llamada a la función requiere paréntesis después del nombre del constructor, aun si no se usan parámetros. Sin embargo, cuando se usan en una declaración, los paréntesis no deben incluirse para un constructor con parámetro cero. Por ejemplo, la declaración `Fecha a();` es incorrecta. La declaración correcta es `Fecha a;`. Sin embargo, cuando se usan parámetros, deben estar encerrados dentro de paréntesis tanto en instrucciones declarativas como en las no declarativas. Los valores de parámetros por omisión deberán incluirse dentro del prototipo del constructor.

El término **constructor por omisión** se usa con bastante frecuencia en C++. Se refiere a cualquier constructor que no requiere ningún argumento cuando es invocado. Esto puede deberse a que no se declararon argumentos, lo cual sucede con el valor por omisión suministrado por el compilador, o debido a que a todos los argumentos se les han dado valores por omisión. Por ejemplo, el constructor `Fecha(int mm = 7, int dd = 4, int aaaa = 2001)` también es un prototipo válido para un constructor por omisión. Aquí, a cada argumento se le ha dado un valor por omisión, y un objeto puede declararse como tipo `Fecha` sin suministrar ningún argumento adicional. Al usar un constructor así, la declaración `Fecha a;` inicializa el objeto `a` con los valores por omisión 7, 4 y 2001.

Para verificar que una función constructora es llamada de manera automática siempre que se crea un objeto nuevo, considérese el programa 9.2. Hay que observar que en la sección de implementación la función constructora usa `cout` para desplegar el mensaje `Se creo un objeto nuevo con valores de datos`. Por tanto, siempre que se invoca al constructor se despliega este mensaje. En vista que la función `main()` crea tres objetos, el constructor es llamado tres veces y el mensaje se despliega tres veces.

Se produce la siguiente salida cuando se ejecuta el programa 9.2:

```
Se creó un objeto nuevo con valores de datos 7, 4, 2005
Se creó un objeto nuevo con valores de datos 7, 4, 2005
Se creó un objeto nuevo con valores de datos 4, 1, 2006
```

Aunque puede usarse cualquier instrucción legítima de C++ dentro de una función constructora, como la instrucción `cout` usada en el programa 9.2, es mejor mantener simples a los constructores y usarlos sólo para propósitos de inicialización. Es necesario agregar un punto más con respecto a la función constructora contenida en el programa 9.2. De acuerdo con las reglas de C++, los miembros del objeto son inicializados en el orden en que son declarados en la sección de declaración de la clase y *no* en el or-

den en que pueden aparecer en la definición de la función dentro de la sección de implementación. Por lo general esto no será un problema, a menos que un miembro sea inicializado usando otro valor de los miembros de datos.

Programa 9.2

```
#include <iostream>
using namespace std;

// sección de declaración de clase

class Fecha
{
  private:
    int mes;
    int dia;
    int anio;
  public:
    Fecha(int = 7, int = 4, int = 2005);   // constructor
};

// sección de implementación

Fecha::Fecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;
  cout << "Se creó un objeto nuevo con valores de datos "
       << mes << ", " << dia << ", " << anio << endl;
}

int main()
{
  Fecha a;            // declara un objeto
  Fecha b;            // declara un objeto
  Fecha c(4,1,2006);  // declara un objeto

  return 0;
}
```

Constructores que llaman

Como se ha visto, los constructores son llamados siempre que se crea un objeto. La declaración real, sin embargo, puede hacerse en una variedad de formas.

Por ejemplo, la declaración

```
Fecha c(4,1,2006);
```

usada en el programa 9.2 también podría haberse escrito como

```
Fecha c = Fecha(4,1,2006);
```

Esta segunda forma declara a `c` como un tipo `Fecha` y luego hace una llamada directa a la función constructora con los argumentos 4, 1 y 2006. Esta segunda forma puede simplificarse cuando sólo se transmite un argumento al constructor. Por ejemplo, si sólo el miembro de datos mes del objeto `c` necesita inicializarse con el valor 8 y los miembros día y año pueden usar los valores por omisión, el objeto puede crearse usando la declaración

```
Fecha c = 8;
```

En vista que ésta se parece a las declaraciones en C, y es una forma más completa usando el signo de igual se conoce como **estilo C de inicialización**. La forma de declaración usada en el programa 9.2 se conoce como **estilo C++ de inicialización**, y es la forma que se usará de manera predominante a lo largo del resto del texto.

Sin importar cuál forma de inicialización use, en ningún caso deberá declararse un objeto con paréntesis vacíos. Por ejemplo, la declaración `Fecha a();` no es la misma que la declaración `Fecha a;`. La última declaración usa los valores del constructor por omisión mientras la primera declaración da como resultado que ningún objeto se cree.

Constructores sobrecargados y en línea

La diferencia principal entre un constructor y otras funciones escritas por el usuario es la forma en que se llama al constructor: los constructores son invocados de manera automática cada vez que se crea un objeto, mientras otras funciones deben ser llamadas de manera explícita por nombre.[9] Como una función, sin embargo, un constructor debe seguir aún todas las reglas aplicables a las funciones escritas por un usuario que se presentaron en el capítulo 6. Esto significa que los constructores pueden tener argumentos por omisión, como se ilustró en el programa 9.1, pueden ser sobrecargados y pueden escribirse como funciones en línea.

Se recordará de la sección 6.1 que sobrecargar a la función permite que se use el mismo nombre de función con diferentes listas de argumentos. Basado en los tipos de argumentos suministrados, el compilador determina cuál función usar cuando se encuentra la llamada. Ahora se verá cómo puede aplicarse a la clase `Fecha`. Por conveniencia se repite a continuación la declaración de clase apropiada:

```
// sección de declaración de clase
class Fecha
{
  private:
    int mes;
```

[9]Esto es cierto para todas las funciones excepto los destructores, los cuales se describen más adelante en esta sección. Una función destructora es llamada de manera automática cada vez que se destruye un objeto.

```
    int dia;
    int anio;
  public:
    Fecha(int = 7, int = 4, int = 2005);   // constructor
};
```

Aquí, el prototipo constructor especifica tres parámetros en número entero, los cuales se usan para inicializar los miembros de datos de mes, día y año.

Un método alternativo para especificar una fecha es usar un número entero largo en la forma año · 10000 + mes · 100 + día. Por ejemplo, la fecha 12/24/1998 usando esta forma es 19981224 y la fecha 2/5/2006 es 20060205.[10] Un prototipo adecuado para un constructor que usa fechas de esta forma es:

```
Fecha(long); // un constructor sobrecargado
```

Aquí, el constructor es declarado como receptor de un argumento en número entero largo. El código para esta nueva función `Fecha`, por supuesto, debe convertir en forma correcta su argumento único en un mes, día y año, y se incluiría dentro de la sección de implementación de la clase. El código real para dicho constructor es:

```
Fecha::Fecha(long aaaammdd) // un segundo constructor
{
  anio = int(aaaammdd/10000.0); // extrae el anio
  mes = int( (aaaammdd - anio * 10000.0) / 100.00 ); // extrae el mes
  dia = int(aaaammdd - anio * 10000.0 - mes * 100.0); // extrae el dia
}
```

No hay que preocuparse demasiado por el código de conversión real usado dentro del cuerpo de la función. El punto importante aquí es el concepto de sobrecargar la función `Fecha()` para proporcionar dos constructores. El programa 9.3 contiene la definición de clase completa dentro del contexto de un programa funcional.

Programa 9.3

```
#include <iostream>
#include <iomanip>
using namespace std;
```

(Continúa)

[10] La razón para especificar fechas de esta manera es que sólo es necesario un número por fecha y que clasificar los números pone de manera automática a las fechas correspondientes en orden cronológico.

(Continuación)

```
// declaración de clase

class Fecha
{
  private:
    int mes;
    int dia;
    int anio;
  public:
    Fecha(int = 7, int = 4, int = 2005);   // constructor
    Fecha(long);              // otro constructor
    void mostrarFecha();      // función miembro para desplegar una fecha
};

// sección de implementación

Fecha::Fecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;
}

Fecha::Fecha(long aaaammdd)
{
  anio = int(aaaammdd/10000.0); // extrae el anio
  mes = int( (aaaammdd - anio * 10000.0)/100.00 ); // extrae el mes
  dia = int(aaaammdd - anio * 10000.0 - mes * 100.0); // extrae el dia
}

void Fecha::mostrarFecha()
{
  cout << "La fecha es ";
  cout << setfill('0')
       << setw(2) << mes << '/'
       << setw(2) << dia << '/'
       << setw(2) << anio % 100; // extrae los últimos dos dígitos del anio
  cout << endl;

  return;
}
```

(Continúa)

(Continuación)

```
int main()
{
  Fecha a, b(4,1,1998), c(20060515L); // declara tres objetos

  a.mostrarFecha();           // despliega los valores del objeto a
  b.mostrarFecha();           // despliega los valores del objeto b
  c.mostrarFecha();           // despliega los valores del objeto c

  return 0;
}
```

La salida proporcionada por el programa 9.3 es:

```
La fecha es 07/04/05
La fecha es 04/01/98
La fecha es 05/15/06
```

Tres objetos son creados en la función `main()` del programa 9.3. El primer objeto, `a`, es inicializado con el constructor por omisión usando sus argumentos por omisión. El objeto `b` también es inicializado con el constructor por omisión pero usa los argumentos 4, 1 y 1998. Por último, el objeto `c`, el cual es inicializado con un número entero largo, usa el segundo constructor en la sección de implementación de la clase. El compilador sabe usar este segundo constructor debido a que el argumento especificado, 20020515L, es designado con claridad como un número entero largo. Vale la pena señalar que ocurriría un error de compilador si ambos constructores `Fecha` tuvieran valores por omisión. En tal caso una declaración como `Fecha d`; sería ambigua para el compilador, por lo que no sería capaz de determinar cuál constructor usar. Por tanto, en cada sección de implementación sólo puede escribirse un constructor por omisión.

Del mismo modo en que los constructores pueden sobrecargarse, también pueden escribirse como funciones en línea. Hacerlo así tan sólo significa definir la función en la sección de declaración de la clase. Por ejemplo, hacer en línea ambos constructores contenidos en el programa 9.3 se logra por la sección de declaración:

```
// declaracion de clase

class Fecha
{
  private:
    int mes;
    int dia;
    int anio;
  public:
    Fecha(int mm = 7, int dd = 4, int aaaa = 2005)
    {
      mes = mm;
      dia = dd;
      anio = aaaa;
    }
```

Punto de información

Funciones de acceso

Una *función de acceso* es cualquier función miembro no constructora que tiene acceso a miembros de datos `private` de una clase. Por ejemplo, la función `mostrarFecha() en la clase Fecha` es una función de acceso. Tales funciones son importantes en extremo porque proporcionan un medio para desplegar valores almacenados en los miembros de datos `private`.

Cuando construya una `clase` asegúrese de proporcionar un conjunto completo de funciones de acceso. Cada función de acceso no tiene que devolver un valor exacto del miembro de datos, pero deberá devolver una representación útil del valor. Por ejemplo, suponga que una fecha como 12/25/2006 es almacenada como una variable de miembro de número entero largo en la forma 20062512. Aunque una función de acceso podría desplegar este valor, por lo general una representación más útil sería 12/25/06 o diciembre 25, 2006.

```
    Fecha(long aaaammdd) // aqui esta el constructor sobrecargado
    {
      anio = int(aaaammdd/10000.0); // extrae el anio
      mes = int( (aaaammdd - anio * 10000.0)/100.00 ); // extrae el mes
      dia = int(aaaammdd - anio * 10000.0 - mes * 100.0); // extrae el dia
    }
};
```

La palabra clave en línea no se requiere en esta declaración debido a que las funciones miembro definidas dentro de la declaración `class` son en línea por omisión.

En general, sólo funciones que pueden codificarse en una sola línea son buenos candidatos para funciones en línea. Esto refuerza la convención que las funciones en línea deberían ser pequeñas. Por tanto, el primer constructor se escribe de manera convencional como

```
Fecha(int mm = 7, int dd = 4, int aaaa = 2005)
  { mes = mm; dia = dd; anio = aaaa; }
```

El segundo constructor, el cual se extiende por tres líneas, no debería escribirse como una función en línea.

Destructores

La contraparte de las funciones constructoras son las funciones destructoras. Los destructores son funciones que tienen el mismo nombre de clase que los constructores, pero precedido por una tilde (~). Por tanto, para la clase `Fecha`, el nombre destructor es `~Fecha()`. Como los constructores, un destructor por omisión que no hace nada es proporcionado por el compilador de C++ en ausencia de un destructor explícito. A diferencia de los constructores, sin embargo, sólo puede haber una función destructora por clase. Esto se debe a que los destructores no utilizan parámetros y tampoco devuelven valores.

Los destructores son llamados de manera automática siempre que un objeto deja de existir y se requieren para "limpiar" los efectos indeseables que pudieran ser dejados por el objeto. Por lo general tales efectos sólo ocurren cuando un objeto contiene un miembro apuntador.

Nota de programación

Métodos mutadores

Un **método mutador**, por lo común llamado mutador, es cualquier método de clase no constructor que cambia los valores de datos de un objeto. Los mutadores se usan para alterar los valores de datos de un objeto después que éste se ha creado y ha sido inicializado de manera automática por un método constructor. Una clase puede contener múltiples mutadores en tanto cada mutador tenga un nombre o lista de parámetros únicos. Por ejemplo, en la clase `Fecha`, podría existir un mutador para cambiar los valores de mes, día y año de un objeto `Fecha`.

Los constructores, cuyo propósito principal es inicializar las variables de miembros de un objeto cuando éste se crea, no se consideran mutadores.

Ejercicios 9.2

1. Determine si los siguientes enunciados son verdaderos o falsos.
   a. Una función constructora debe tener el mismo nombre que su clase.
   b. Una clase sólo puede tener una función constructora.
   c. Una clase sólo puede tener una función constructora por omisión.
   d. Un constructor por omisión sólo puede ser suministrado por el compilador.
   e. Un constructor por omisión puede no tener parámetros o todos los parámetros deben tener valores por omisión.
   f. Debe declararse un constructor para cada `class`.
   g. Debe declararse un constructor con un tipo para devolver.
   h. Se llama de manera automática un constructor cada vez que se crea un objeto.
   i. Una clase sólo puede tener una función destructora.
   j. Un destructor debe tener el mismo nombre que su clase, precedido por una tilde (~).
   k. Un destructor puede tener argumentos por omisión.
   l. Debe declararse un destructor para cada clase.
   m. Debe declararse un destructor con un tipo para devolver.
   n. Un destructor es llamado de manera automática cada vez que un objeto deja de existir.
   o. Los destructores no son útiles cuando la clase contiene un miembro de datos apuntador.

2. Para el programa 9.3, ¿qué fecha se inicializaría para el objeto `c` si se usara la declaración `Fecha c(15);` en lugar de la declaración `Fecha c(20060515L);`?

3. Modifique el programa 9.3 de modo que el único miembro de datos de la clase sea un número entero largo llamado `aaaammdd`. Haga esto sustituyendo la declaración

```
long aaaammdd;
```

por las declaraciones existentes

```
int mes;
int dia;
int anio;
```

Luego, usando los mismos prototipos de la función constructora declarados en la actualidad en la sección de declaración de la clase, vuelva a escribirlos de modo que la función `Fecha(long)` se vuelva el constructor por omisión y la función `Fecha(int, int, int)` convierta un mes, día y año en la forma apropiada para el miembro de datos de la clase.

4. **a.** Construya una clase `Tiempo` que contenga los miembros de datos enteros segundos, minutos y horas. Haga que la clase contenga dos constructores: el primero deberá ser un constructor por omisión que tenga el prototipo `tiempo(int, int, int)`, el cual usa valores por omisión de 0 para cada miembro de datos. El segundo constructor deberá aceptar un número entero largo que represente un número total de segundos y desensamble el número entero largo en `horas`, `minutos` y `segundos`. El miembro final de la función deberá desplegar los miembros de la clase de datos.
   b. Incluya la clase escrita para el ejercicio 4a dentro del contexto de un programa completo.
5. **a.** Construya una clase llamada `Estudiante` consistente de un número entero de identificación del estudiante, un arreglo de cinco calificaciones en número de precisión doble y un número entero que represente el número total de calificaciones introducidas. El constructor para esta clase deberá inicializar todos los miembros de datos de Estudiante en cero. Deberán incluirse en la clase funciones miembro para 1) introducir un número de identificación del estudiante, 2) introducir una sola calificación de prueba y actualizar el número total de calificaciones introducidas y 3) calcular una calificación promedio y desplegar la identificación del estudiante seguida por la calificación promedio.
   b. Incluya la clase construida en el ejercicio 5a dentro del contexto de un programa completo. Su programa deberá declarar dos objetos del tipo `Estudiante` y aceptar y desplegar datos para los dos objetos para verificar la operación de las funciones miembro.
6. **a.** En el ejercicio 4 se le pidió que construyera una clase `Tiempo`. Para dicha clase incluya una función `tic()` que incremente el tiempo por un segundo. Pruebe su función para asegurar que se incrementa en forma correcta en un nuevo minuto y una nueva hora.
   b. Modifique la clase `Tiempo` escrita para el ejercicio 6a para incluir una función `destic()` que disminuya el tiempo por un segundo. Pruebe su función para asegurar que se disminuye en forma correcta el tiempo a una hora anterior y a un minuto anterior.

9.3 APLICACIONES

Ahora que ha comprendido cómo están construidas las clases y la terminología usada para describirlas, se aplicará este conocimiento a dos aplicaciones nuevas. En la primera aplicación se desarrollará una clase para determinar el área del piso de una habitación con forma rectangular. En la segunda aplicación se simulará la operación de una bomba de gasolina.

Aplicación 1: Construir un objeto de habitación

En esta aplicación, se creará una clase a partir de la cual puedan construirse objetos tipo habitación. El área del piso de la habitación debe calcularse para cualquier tamaño de ha-

bitación cuando se conocen su largo y ancho. Para propósitos de modelado, supóngase que todas las habitaciones son rectangulares.

Paso 1 *Analizar el problema*

En esta aplicación, sólo tenemos un tipo de objeto, el cual es una habitación de forma rectangular. Debido a esto, el piso de la habitación puede ser indicado por su largo y ancho. Una vez que estos atributos han sido asignados para una habitación, el área de su piso puede calcularse como el largo de la habitación multiplicado por su ancho.

Paso 2 *Desarrollar una solución*

Para esta aplicación, el largo y ancho de una habitación son los únicos atributos de interés. Éstos pueden representarse mediante variables en números de precisión doble que se llamarán largo y ancho. El servicio requerido de la clase es un constructor para establecer los atributos de largo y ancho de una habitación, una función de acceso para desplegar los valores de atributo de la habitación, un mutador para cambiar los valores de los atributos de una habitación y una función para determinar el área del piso a partir de sus valores de largo y ancho. De manera arbitraria se nombrará a la clase `tipoHab`, a la función de acceso mostrar`ValHab()`, a la función mutadora `estValNueHab()` y a la función de cálculo del área `calcularAreaHab()`.

Paso 3 *Codificar la clase*

A partir del diseño y la elección de los nombres de los atributos y las funciones de la clase, una declaración de clase adecuada es la siguiente:

```
class tipoHab
{
  // sección de declaración de datos
  private:
    double largo;  // declara largo como una variable doble
    double ancho;  // declara ancho como una variable doble

  public:
    tipoHab(); // la instrucción de declaración del constructor
    void mostrarValHab();
    void estValNueHab();
    void calcularAreaHab();
};
```

Se han declarado dos miembros de datos, `largo` y `ancho`, y cuatro funciones de clase. Los miembros de datos largo y ancho almacenarán el largo y el ancho de una habitación, respectivamente. Los servicios proporcionados por la clase son un constructor para crear un objeto habitación, una función de acceso para desplegar los valores de largo y ancho de un objeto habitación, un mutador para cambiar los valores de largo y ancho de una habitación y, por último, una función de cálculo para calcular el área del piso de una habitación. Para lograr estos servicios, una sección de implementación de clase adecuada es la siguiente:

```
// sección de implementación de métodos
tipoHab::tipoHab(double l, double a) // éste es un constructor
{
  largo = l;
```

```
  ancho = a;
  cout << "Se ha creado un objeto habitación nuevo usando el constructor
          por omisión.\n\n";
}

void tipoHab::mostrarValHab()  // ésta es una función de acceso
{
  cout << " largo = " << largo
       << "\n ancho = " << ancho << endl;
}

void tipoHab::estValNueHab(double l, double a) // éste es un mutador
{
  largo = l;
  ancho = a;
}

void tipoHab::calcularAreaHab() // ésta realiza un cálculo
{
  cout << (largo * ancho);
}
```

Cada una de estas funciones es sencilla. Cuando se declara un objeto habitación, se inicializará con un largo y ancho de cero a menos que se proporcionen valores específicos en la declaración. La función de acceso despliega los valores almacenados en largo y ancho, y el mutador permite reasignar valores después que se ha creado un objeto habitación. Por último, la función de cálculo despliega el área de una habitación al multiplicar su largo por su ancho.

Paso 4 Probar y corregir el programa

Probar la clase `tipoHab` implica probar cada función de la clase. Para hacer esto, se incluye la clase `tipoHab` dentro del contexto de un programa funcional, el cual se enlista como programa 9.4.

Programa 9.4

```
#include <iostream>
using namespace std;
class tipoHab
{
  // sección de declaración de datos
  private:
    double largo;   // declara largo como una variable doble
    double ancho;   // declara ancho como una variable doble
```

(Continúa)

(Continuación)

```
  public:
    tipoHab(double = 0.0, double = 0.0); // la instrucción de declaración
del constructor
    void mostrarValHab();
    void estValNueHab(double, double);
    void calcularAreaHab();
};

// seccion de implementación de métodos
tipoHab::tipoHab(double l, double a) // éste es un constructor
{
  largo = l;
  ancho = a;
  cout << "Se ha creado un objeto habitación nuevo usando el constructor
  por omision.\n\n";
}

void tipoHab::mostrarValHab()   // ésta es una función de acceso
{
  cout << " largo = " << largo
       << "\n ancho = " << ancho << endl;
}

void tipoHab::estValNueHab(double l, double a) // éste es un mutador
{
  largo = l;
  ancho = a;
}

void tipoHab::calcularAreaHab() // ésta realiza un cálculo
{
  cout << (largo * ancho);
}
```

```
int main()
{
  tipoHab habUno(12.5, 18.2);   // declara una variable de tipo tipoHab

  cout << "Los valores para esta habitación son: \n";
  habUno.mostrarValHab();       // usa un método de clase en este objeto
  cout << "\nEl area del piso de esta habitación es: ";
  habUno.calcularAreaHab();     // usa otro método de clase en este objeto

  habUno.estValNueHab(5.5, 9.3); // llama al mutador

  cout << "\n\nLos valores para esta habitación han sido cambiados a: \n";
  habUno.mostrarValHab();
```

(Continúa)

(Continuación)

```
    cout << "\nEl area del piso de esta habitacion es: ";
    habUno.calcularAreaHab();

    cout << endl;
    return 0;
}
```

La porción sombreada en gris más claro del programa 9.4 contiene la construcción de clase que ya se ha descrito. Para ver cómo se usa esta clase, concéntrese en la sección más oscura del programa que contiene la función `main()`. Esta función crea un objeto de habitación que tiene un largo de 12.5 y un ancho de 18.2. Las dimensiones de esta habitación se despliegan usando la función `mostrarValHab()`, y el área se calcula y despliega usando la función `calcularAreaHab()`. Las dimensiones de la habitación se reinician, se despliegan y el área de la habitación se recalcula. La salida producida por el programa 9.4 es la siguiente:

```
Se ha creado un objeto habitación nuevo usando el cons-
tructor por omisión.

Los valores para esta habitación son:
   largo = 12.5
   ancho = 18.2

El area del piso de esta habitación es: 227.5

Los valores para esta habitación han sido cambiados a:
   largo = 5.5
   ancho = 9.3

El área del piso de esta habitación es: 51.15
```

Los requisitos básicos de la programación orientada a objetos son evidentes incluso en un programa tan simple como el programa 9.4. Antes que pueda escribirse la función `main()`, se debe construir una clase útil. Esto es típico de los programas que usan objetos. Para tales programas el proceso de diseño tiene la carga inicial de dar una consideración cuidadosa a la clase, su declaración e implementación. El código contenido en la sección de implementación elimina de manera efectiva el código que de otra manera debería ser parte del trabjo de `main()`. Por tanto, cualquier programa que use el objeto no tiene que repetir los detalles de implementación dentro de su función `main()`. Más bien, la función `main()` y cualquier función llamada por `main()` sólo se ocupa por en enviar mensajes a sus objetos para activarlos de manera apropiada. La forma en que responde el objeto a los mensajes y la forma en que se conserva el estado del objeto no es de interés para `main()`; estos detalles están ocultos dentro de la construcción de `class`.

Aplicación 2: Simulación de una bomba de gasolina con un objeto simple

En esta sección, se presenta la primera clase necesaria para una simulación que requiere dos clases separadas. La primera clase, que es la que se desarrolla en esta sección, modela una bomba de gasolina. La segunda clase, la cual completa la simulación y se desarrolla en la sección 10.5, modela la llegada de múltiples clientes, cada uno con varias solicitudes por diferentes cantidades de gasolina que se han de surtir. La simulación completa se basa en el siguiente requerimiento:

Se ha solicitado que escribamos un programa que simule la operación de una bomba de gasolina. En cualquier momento durante la simulación, deberíamos poder determinar, a partir de la bomba, el precio por galón de gasolina y la cantidad restante en el tanque de suministro. Si la cantidad de gasolina en el tanque de suministro es mayor que o igual a la cantidad de gasolina solicitada, la solicitud debe despacharse; de lo contrario, sólo deberá usarse la cantidad disponible en el tanque de suministro. Una vez que se ha bombeado la gasolina, deberá desplegarse el precio total de los galones surtidos y la cantidad de gasolina en galones que se ha bombeado deberá restarse de la cantidad sobrante en el tanque de suministro.

Para la simulación, suponga que la bomba está inactiva en forma aleatoria entre 1 a 15 minutos entre llegadas de clientes y que un cliente solicita en forma aleatoria entre 3 y 20 galones de gasolina. Aunque la capacidad del tanque de suministro por omisión es de 500 galones, suponga que la cantidad inicial de gasolina en el tanque para esta simulación es de sólo 300 galones. Al principio, el programa deberá simular un lapso de media hora.

Además, para cada llegada y solicitud de gasolina, se desea saber el tiempo de inactividad antes de la llegada del cliente, cuántos galones de gasolina fueron surtidos y el precio total de la transacción. La bomba en sí deberá registrar el precio por galón de gasolina y la cantidad de gasolina restante en el tanque de suministro. Por lo general, el precio por galón es $1.80, pero el precio para la simulación deberá ser $2.00.

Para esta parte de la simulación, se construirá una clase de bomba de gasolina que pueda utilizarse en la simulación final, la cual se completa en la sección 10.5.

Paso 1 Analizar el problema

Este problema implica dos tipos de objetos distintos. El primero es una persona que puede llegar de manera aleatoria entre 1 y 15 minutos y puede solicitar de manera aleatoria entre 3 y 20 galones de gasolina. El segundo tipo de objeto es la bomba de gasolina. Para esta parte del problema, la meta será crear una clase de bomba de gasolina adecuada que pueda usarse en la simulación final, la cual se completa en la siguiente aplicación.

El modelo para construir una clase de bomba de gasolina que cumpla con los requisitos de la simulación se describe en seudocódigo como el siguiente:

```
Poner Bomba en servicio
   Inicializar la cantidad de gasolina en el tanque de suministro
   Inicializar el precio por galón de gasolina


Desplegar valores
   Desplegar la cantidad de gasolina en el tanque de suministro
   Desplegar el precio por galón
```

Despachar una cantidad de gasolina
Si la cantidad en el tanque de suministro es mayor que o igual a la cantidad solicitada
Establecer la cantidad surtida igual a la cantidad solicitada
De no ser así
Establecer la cantidad surtida igual a la cantidad en el tanque de suministro
Termina el si
Restar la cantidad surtida de la cantidad en el tanque de suministro
Calcular el precio total como el precio por galón por la cantidad surtida
Desplegar los galones de gasolina solicitados
Desplegar los galones de gasolina despachados
Desplegar la cantidad restante en el tanque de suministro
Desplegar el precio total para la cantidad de gasolina despachada

Paso 2 Desarrollar una solución

A partir de la descripción en seudocódigo, la implementación de una clase `Bomba` es sencilla. Los atributos de interés requeridos para la bomba son la cantidad de galones en el tanque de suministro y el precio por galón. Las operaciones requeridas incluyen suministrar valores iniciales para los atributos de la bomba, consultar los valores de atributo a la bomba y satisfacer una solicitud de gasolina.

Debido a que los dos atributos, la cantidad en el tanque y el precio por galón, pueden tener valores fraccionarios, es conveniente hacerlos valores de precisión doble. Además, necesitan proporcionarse tres servicios. El primero consiste en inicializar los atributos de una bomba, lo cual consiste en establecer valores para la cantidad en el tanque de suministro y el precio por galón. El segundo consiste en satisfacer una solicitud de gasolina, y el tercer servicio proporciona una lectura de los valores de atributo actuales de la bomba.

Paso 3 Codificar la solución

Una declaración de clase adecuada es la siguiente:

```
const double CANTIDAD_EN_TANQUE = 500; // galones iniciales en el tanque
const double PRECIO_OMISION = 1.80; // precio por galón

class Bomba
{
  // sección de declaración de datos
  private:
    double cantEnTanque;
    double precio;

  // declaraciones de métodos
  public:
    Bomba(double = PRECIO_OMISION, double = CANTIDAD_EN_TANQUE); // cons
                  tructor
    void obtenerValores();
    void solicitud(double);
};
```

Analicemos esta declaración de clase examinado de manera individual sus miembros de datos y métodos. Primero, hay que observar que se han declarado dos constantes simbólicas y dos variables de instancia `private`. Como miembros `private`, sólo puede tenerse acceso a estos atributos de datos por medio de los métodos de miembro de la clase: `Bomba(), obtenerValores()` y `solicitud()`. Son estas funciones las que proporcionan los servicios externos disponibles para cada objeto `Bomba`.

La función `Bomba()`, la cual tiene el mismo nombre que su clase, es la función constructora que es invocada de manera automática cuando se crea un objeto de tipo `Bomba`. La función `obtenerValores()` proporciona una lectura de los valores de atributos actuales y la función `solicitud()` maneja la lógica de satisfacer la solicitud de gasolina de un cliente. Para lograr estos servicios, una sección de implementación de clase adecuada es la siguiente:

```
// sección de implementación de métodos

Bomba::Bomba(double precioHoy, double cantidadEnTanque)
{
  cantEnTanque = cantidadEnTanque;
  precio = precioHoy;
}

void Bomba::obtenerValores()
{
  cout << "El tanque de gasolina tiene " << cantEnTanque << " galones de
          gasolina." << endl;
  cout << "El precio por galón de gasolina es $" << setiosflags
          (ios::showpoint)
       << setprecision(2) << setiosflags(ios::fixed) << precio << endl;
}

void Bomba::solicitud(double cantSurt)
{
  double surtida;

  if (cantEnTanque >= cantSurt)
     surtida = cantSurt;
  else
     surtida = cantEnTanque;

  cantEnTanque -= surtida;
  cout << cantSurt << " galones fueron solicitados " << endl;
  cout << surtida << " galones fueron surtidos" << endl;
  cout << cantEnTanque << " galones quedan en el tanque" << endl;
  cout << "El precio total es $" << setiosflags(ios::showpoint)
       << setprecision(2) << (surtida * precio) << endl;

  return;
}
```

La función constructora es sencilla. Cuando se declara un objeto `Bomba`, se inicializará con una cantidad determinada de gasolina en el tanque de suministro y un precio determinado por galón. Si no se dan valores, se usan los valores por omisión de $1.80 por galón y 500 galones; si sólo se proporciona el precio por galón, el constructor usa el valor por omisión de 500 para el segundo argumento faltante.

La función `obtenerValores()` definida en la sección de implementación proporciona una lectura de los valores de atributo actuales. La función `solicitud()` es la que es más complicada debido a que proporciona el servicio de bombeo principal. El código que sigue a los requerimientos de la bomba proporciona toda la gasolina solicitada a menos que la cantidad restante en el tanque de suministro sea menor que la cantidad solicitada. Por último, resta la cantidad surtida de la cantidad en el tanque y calcula el valor total en dólares de la transacción.

Paso 4 Probar y corregir la clase

Probar la clase `Bomba` requiere probar cada función de la clase. Para hacer esto, considere el programa 9.5.

Programa 9.5

```
#include <c:\\cpcode\\Bomba.cpp>

int main()
{
  Bomba a(2.00, 300), b;   // declara 2 objetos del tipo Bomba

  a.obtenerValores();
  cout << endl;
  a.solicitud(20.0);
  cout << endl;
  a.solicitud(290.0);
  b.obtenerValores();

  return 0;
}
```

En el programa 9.5, se ha incluido la clase `Bomba` usando la siguiente instrucción:

```
#include <c:\\cpcode\\Bomba.cpp>
```

Esto supone que la clase `Bomba` reside en la carpeta llamada `cpcode` en la unidad C, y está guardada como el archivo llamado `Bomba.cpp`. Una instrucción equivalente es la siguiente:

```
#include "c:\\cpcode\\Bomba.cpp"
```

En ambas instrucciones include se requieren las diagonales invertidas dobles porque una sola diagonal invertida se interpretaría como un carácter de escape.

Punto de información

Encapsulamiento

El término **encapsulamiento** se refiere a empacar diversos elementos en una sola unidad. Por ejemplo, una función se usa para encapsular los detalles de un algoritmo. Del mismo modo, una clase encapsula tanto a variables como a funciones en un solo paquete.

Aunque el término encapsulamiento se usa a veces para referirse al proceso de ocultamiento de información, este uso no es preciso desde el punto de vista técnico. La relación correcta entre términos es que el ocultamiento de información se refiere al encapsulamiento y ocultamiento de todos los detalles de la implementación.

Dentro del método `main()` se incluyen ocho instrucciones. La primera instrucción crea un objeto de tipo `Bomba`. El tanque de suministro para el primer objeto `Bomba` contiene 300 galones y el precio por galón se establece en $2.00; el segundo objeto `Bomba` usa los valores por omisión CANTIDAD_EN_TANQUE y PRECIO_OMISION, los cuales son 500 y 1.80, respectivamente.

Se hace una llamada a `obtenerValores()` para desplegar los valores de atributos de la primera bomba. La siguiente instrucción es una solicitud de 20 galones de gasolina del primer objeto `Bomba`. Esto es seguido por una solicitud de 290 galones, lo cual excede la gasolina restante en el tanque de suministro. Por último, se despliegan los valores de atributo para la segunda bomba. La salida producida por el programa 9.5 es la siguiente:

```
El tanque de gasolina tiene 300 galones de gasolina.
El precio por galon de gasolina es $2.00

20.00 galones fueron solicitados
20.00 galones fueron surtidos
280.00 galones quedan en el tanque
El precio total es $40.00

290.00 galones fueron solicitados
280.00 galones fueron surtidos
0.00 galones quedan en el tanque
El precio total es $560.00

El tanque de gasolina tiene 500 galones de gasolina.
El precio por galón de gasolina es $1.80
```

Como lo indica esta salida, todos los métodos de la clase `Bomba` proporcionan la funcionalidad correcta. De manera específica, ambos objetos `Bomba` son inicializados en forma correcta por el constructor, y el método `solicitud()` suministra la cantidad de gasolina solicitada al precio correcto hasta que el tanque de suministro se ha vaciado.

Ejercicios 9.3

1. Introduzca el programa 9.4 en su computadora y ejecútelo.
2. Modifique la función `main()` en el programa 9.4 para crear una segunda habitación que tenga un largo de 9 y un ancho de 12. Haga que el programa calcule el área de esta nueva habitación.

3. **a.** Modifique la función `main()` en el programa 9.4 para crear cuatro habitaciones: vestíbulo, cocina, comedor y sala. Las dimensiones de estas habitaciones son las siguientes:

```
Vestibulo: largo = 12.40, ancho = 3.5
Cocina: largo = 14, ancho = 14
Sala: largo = 12.4, ancho = 20
Comedor: largo = 14, ancho = 9.5
```

   Su programa deberá desplegar el área de cada habitación y el área total de las cuatro habitaciones combinadas.

   b. El área total de todas las habitaciones puede calcularse y guardarse usando una variable de clase. Para hacer esto, ¿qué tipo de variable piensa que tendría que ser esta variable de clase?

4. **a.** Modifique la función `main()` en el programa 9.5 para usar un ciclo `while` que llame a la función de solicitud de `Bomba` con un número aleatorio entre 3 y 20. El ciclo `while` deberá terminar después que se hayan hecho cinco solicitudes.

   b. Modifique la función `main()` escrita para el ejercicio 4a para proporcionar una simulación de 30 minutos de la operación de la bomba de gasolina. Para hacer esto tendrá que modificar el ciclo `while` para seleccionar un número aleatorio entre 1 y 15 que represente el tiempo de inactividad entre solicitudes de clientes. Haga que la simulación se detenga una vez que el tiempo de inactividad exceda los 30 minutos.

5. **a.** Construya una definición de clase de un tipo de objeto `Persona`. La clase no tendrá atributos, una sola función constructora y dos funciones miembro adicionales llamadas `llegada()` y `galones()`. La función constructora deberá llamar tan sólo a `srand()` con el argumento tiempo(`NULL`) para inicializar la función `rand()`. La función `llegada()` deberá proporcionar un número aleatorio entre 1 y 15 como valor a devolver, mientras la función `galones()` deberá proporcionar un número aleatorio entre 3 y 20.

   b. Pruebe las funciones de la clase `Persona` escritas para el ejercicio 5a en un programa funcional completo.

   c. Use la función de la clase `Persona` para simular una llegada aleatoria de una `Persona` y una solicitud aleatoria de galones de gasolina dentro del programa escrito para el ejercicio 4b.

6. Modifique el programa 9.5 de modo que la definición de clase `Bomba` resida en un archivo llamado `Bomba.h`. Luego haga que el programa 9.5 use una instrucción `#include` para incluir la definición `class` dentro del programa. Asegúrese de usar un nombre de ruta completo en la instrucción `#include`. Por ejemplo, si `Bomba.h` reside en el directorio llamado `foo` en la unidad C, la instrucción `include` deberá ser

```
#include <c:\\foo\\Bomba.h>.
```

7. Construya una clase llamada Luz que simule un semáforo. El atributo de color de la clase deberá cambiar de `Verde` a `Amarillo` a `Rojo` y luego de vuelta a `Verde` por la función `cambio()` de la clase. Cuando se crea un objeto Luz nuevo su color inicial deberá ser `Rojo`.

8. a. Construya una definición de clase que pueda usarse para representar a un empleado de una compañía. Cada empleado es definido por un número entero de identificación, una tasa salarial en punto flotante y el número máximo de horas que el empleado debería trabajar cada semana. Los servicios de la clase deberán proporcionar la capacidad para introducir datos, cambiar y datos desplegar los datos existentes para un empleado nuevo.

b. Incluya la definición de clase creada para el ejercicio 8a en un programa en C++ funcional que le pida al usuario que introduzca datos para tres empleados y despliegue los datos introducidos.

c. Modifique el programa escrito para el ejercicio 8b para incluir un menú que ofrezca al usuario las siguientes opciones:

1. Agregar un empleado
2. Modificar los datos del empleado
3. Eliminar a un empleado
4. Salir de este menú

En respuesta a una opción el programa deberá iniciar la acción apropiada para implementar la opción.

9. a. Construya una definición de clase que pueda usarse para representar tipos de alimentos. Un tipo de alimento se clasifica como básico o preparado. Los alimentos básicos se clasifican además como `Lacteos`, `Carnes`, `Frutas`, `Vegetales` o `Cereales`. Los servicios proporcionados por la clase deberán ser la capacidad para introducir datos para un alimento nuevo, la capacidad para cambiar datos para un alimento nuevo y la capacidad para desplegar los datos existentes para un alimento nuevo.

b. Incluya la definición de clase creada para el ejercicio 9a en un programa en C++ funcional que pida al usuario que introduzca datos para cuatro opciones de alimentos y despliegue los datos introducidos.

c. Modifique el programa escrito para el ejercicio 9b para incluir un menú que ofrezca al usuario las siguientes opciones:

1. Agregar una opción de alimento
2. Modificar una opción de alimento
3. Eliminar una opción de alimento
4. Salir de este menú

En respuesta a una opción el programa deberá iniciar la acción apropiada para implementar la opción.

9.4 Errores comunes de programación

Los errores de programación más comunes asociados inicialmente con la construcción de clases son

1. Olvidar terminar la sección de declaración de clase con un punto y coma.
2. Incluir un tipo a devolver con el prototipo del constructor o dejar de incluir un tipo a devolver con los otros prototipos de funciones.

3. Usar el mismo nombre para un miembro de datos que para una función miembro.
4. Definir más de un constructor por omisión para una clase.
5. Olvidar incluir el nombre de la clase y el operador de alcance, `::`, en la línea de encabezado de todas las funciones miembro definidas en la sección de implementación de clase.

Todos estos errores producirán un mensaje de error del compilador.

9.5 RESUMEN DEL CAPÍTULO

1. Una **clase** es un tipo de datos definido por el programador. Los **objetos** de una clase pueden definirse y tener la misma relación con su clase que las variables con los tipos de datos integrados de C++.
2. Una definición de clase consiste en una sección de declaración y una de implementación. La forma más común de una definición de clase es

```
// sección de declaración de clase
class nombre
{
  private:
   una lista de declaraciones de variables;
  public:
   una lista de prototipos de funciones;
};

// sección de implementación de la clase
definiciones de funciones de la clase
```

   Las variables y funciones declaradas en la sección de declaración de clase se conocen de manera colectiva como **miembros de clase**. Las variables se conocen de manera individual como miembros de datos de clase y las funciones como funciones miembro de clase. Los términos `private` y `public` son especificadores de acceso. Una vez que se ha enlistado el especificador de acceso, permanece en vigor hasta que se da otro especificador de acceso. La palabra clave `private` especifica que los miembros de clase que siguen son privados de la clase y sólo pueden tener acceso a ellos las funciones miembros. La palabra clave `public` especifica que se puede tener acceso a los miembros de la clase que siguen desde fuera de la clase. En general, todos los miembros de datos deberían especificarse como `private` y todas las funciones como `public`.
3. Las funciones de clase enlistadas en la sección de declaración pueden escribirse en línea o incluir sus definiciones en la sección de implementación de clase. Excepto por las funciones constructoras y destructoras, todas las funciones de clase definidas en la sección de implementación de la clase tienen la forma de línea de encabezado

   tipoDevuelto nombreClase::nombreFunción(lista de parámetros);

Excepto por la adición del nombre de la clase y el operador de alcance, `::`, los cuales se requieren para identificar el nombre de la función con la clase, esta línea de encabezado es idéntica a la línea de encabezado usada para cualquier función escrita por el usuario.

4. Una **función constructora** es una función especial que se llama de manera automática cada vez que se declara un objeto. Debe tener el mismo nombre que su clase y no puede tener ningún tipo a devolver. Su propósito es inicializar cada objeto declarado.
5. Si no se declara un constructor para una clase el compilador suministrará un constructor por omisión. Ésta es una función que no hace nada que tiene la definición
   `nombreClase::nombreClase(void){}`.
6. El término **constructor por omisión** se refiere a cualquier constructor que no requiere ningún argumento cuando es invocado. Esto puede deberse a que no se declaran parámetros (como es el caso para el constructor por omisión suministrado por el compilador) o debido a que a todos los parámetros se les han dado valores por omisión.
7. Cada clase sólo puede tener un constructor por omisión. Si se define cualquier constructor definido por el usuario el compilador no creará su constructor por omisión.
8. Los objetos se crean usando un estilo C++ o un estilo C de declaración. El estilo C++ de declaración tiene la forma:
   nombreClase lista-de-nombres-de-objeto(lista de inicializadores);
   donde la lista de inicializadores es opcional. Un ejemplo de este estilo de declaración, incluyendo inicializadores, para una clase llamada `Fecha` es
   ```
   Fecha a, b, c(12,25,2006);
   ```
   Aquí se declara que los objetos `a` y `b` son del tipo `Fecha` y se inicializan usando el constructor por omisión mientras el objeto `c` es inicializado con los valores 12, 25 y 2006.
   El estilo C de declaración equivalente, incluyendo la lista opcional de inicializadores, tiene la forma
   nombreClase nombreObjeto = nombreClase(lista de inicializadores);
   Un ejemplo de este estilo de declaración para una clase llamada `Fecha` es
   ```
   Fecha c = Fecha(12,25,2006);
   ```
   Aquí el objeto `c` se crea e inicializa con los valores 12, 25 y 2006.
9. Los constructores pueden sobrecargarse de la misma manera que otras funciones en C++ escritas por el usuario.
10. Si se define un constructor para una clase, también deberá escribirse un constructor por omisión definido por el usuario, ya que el compilador no lo suministrará.
11. Una función destructora se llama cada vez que un objeto sale del alcance. Los destructores deben tener el mismo nombre que su clase, pero precedido con una tilde (~). Sólo puede haber un destructor por clase.
12. Una **función destructora** no toma argumentos y no devuelve ningún valor. Si no se incluye un destructor definido por el usuario en una clase el compilador proporcionará un destructor que no hace nada.

CAPÍTULO 10
Funciones de clases y conversiones

TEMAS

10.1 ASIGNACIÓN
- CONSTRUCTORES DE COPIA
- INICIALIZACIÓN BASE/MIEMBRO

10.2 CARACTERÍSTICAS ADICIONALES DE LA CLASE
- ALCANCE DE LA CLASE
- MIEMBROS DE CLASE TIPO STATIC
- FUNCIONES AMIGAS

10.3 FUNCIONES OPERADOR
- FUNCIONES OPERADOR COMO AMIGAS

10.4 CONVERSIONES DEL TIPO DE DATOS
- CONVERSIÓN DE INTEGRADO A INTEGRADO
- CONVERSIÓN DE INTEGRADO A CLASE
- CONVERSIÓN DE CLASE A INTEGRADO
- CONVERSIÓN DE CLASE A CLASE

10.5 APLICACIÓN: SIMULACIÓN DE UNA BOMBA DE GASOLINA CON OBJETOS MÚLTIPLES

10.6 HERENCIA DE CLASE
- ESPECIFICACIONES DE ACCESO
- UN EJEMPLO

10.7 POLIMORFISMO

10.8 ERRORES COMUNES DE PROGRAMACIÓN

10.9 RESUMEN DEL CAPÍTULO

La creación de una clase requiere que se proporcione la capacidad para declarar, inicializar, asignar, manipular y desplegar miembros de datos. En el capítulo anterior se presentó la declaración, inicialización y despliegue de objetos. En este capítulo se continuará la construcción de clases proporcionando los mecanismos para crear capacidades de operador y conversión similares a las inherentes en los tipos integrados de C++. Con estas adiciones los tipos definidos por el usuario tendrán toda la funcionalidad de los tipos integrados.

10.1 ASIGNACIÓN

En el capítulo 3 se vio cómo el operador de asignación de C++, =, ejecuta la asignación entre variables. En esta sección se verá cómo funciona la asignación cuando se aplica a objetos y cómo definir el operador de asignación para cancelar el valor por omisión proporcionado para las clases definidas por los usuarios.

Para implementar un ejemplo de asignación específico, considérese la función `main()` del programa 10.1. Observe que la sección de implementación de la clase `Fecha` en el programa 10.1 no contiene función de asignación. No obstante, se esperaría que la instrucción de asignación `a = b;` en `main()` asignara los valores de miembros de datos de `b` a sus contrapartes en `a`. Éste es, de hecho, el caso y es verificado por la salida producida cuando se ejecuta el programa 10.1:

```
La fecha almacenada en a originalmente es 04/01/07
Despues de la asignacion la fecha almacenada en a es
12/18/08
```

El tipo de asignación ilustrado en el programa 10.1 se conoce como ***asignación miembro por miembro***. En ausencia de cualesquiera instrucciones específicas en contrario, el compilador de C++ construye este tipo de operador de asignación por omisión para cada clase. Si la clase *no* contiene ningún apuntador a miembros de datos este operador de asignación por omisión es adecuado y puede usarse sin mayor consideración. Antes de abordar los problemas que pueden ocurrir con los apuntadores a miembros de datos, se verá cómo construir operadores de asignación explícitos propios.

Los operadores de asignación, como todos los miembros de clase, se declaran en la sección de declaración de la clase y se definen en la sección de implementación de la clase. Para la declaración de operadores, sin embargo, debe incluirse la palabra clave `operator` en la declaración. Usando esta palabra clave, una declaración ***simple de operador*** de asignación tiene la forma:

```
void operator=(NombreClase&);
```

Aquí la palabra clave `void` indica que la asignación no devuelve ningún valor, `operator=` indica que se está sobrecargando el operador de asignación con nuestra propia versión, y el nombre de la clase y el signo ampersand dentro de los paréntesis indican que el argumento para `operator` es una referencia a la clase. Por ejemplo, para declarar un operador de asignación simple para la clase `Fecha`, puede usarse la declaración:

```
void operator=(Fecha&);
```

puede usarse.

La implementación real del operador de asignación se define en la sección de implementación. Para nuestra declaración, una implementación adecuada es

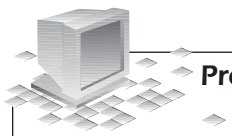
```
void Fecha::operator=(Fecha& fechanueva)
{
  dia = fechanueva.anio;      // asigna el día
  mes = fechanueva.mes;      // asigna el mes
  anio = fechanueva.anio;    // asigna el año
}
```

Programa 10.1

```
#include <iostream>
#include <iomanip>
using namespace std;

// declaracion de clase
class Fecha
{
  private:
    int mes;
    int dia;
    int anio;
  public:
    Fecha(int = 7, int = 4, int = 2005);   // constructor
    void mostrarFecha();   // función miembro para desplegar una fecha
};

// sección de implementación
Fecha::Fecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;
}

void Fecha::mostrarFecha()
{
  cout << setfill('0')
       << setw(2) << mes << '/'
       << setw(2) << dia << '/'
       << setw(2) << anio % 100;
  return;
}

int main()
{
  Fecha a(4,1,2007), b(12,18,2008); // declara dos objetos

  cout << "\nLa fecha almacenada en a originalmente es ";
  a.mostrarFecha(); // despliega la fecha original
  a = b;            // asigna el valor de b a a
  cout << "\nDespués de la asignación la fecha almacenada en a es ";
  a.mostrarFecha(); // despliega los valores de a
  cout << endl;
  return 0;
}
```

El uso del argumento de referencia en la definición de esta operación no es accidental. De hecho, una de las razones principales para agregar variables de referencia a C++ fue facilitar la construcción de operadores sobrecargados y hacer más natural la notación.[1] En esta definición `fechanueva` se define como una referencia a una clase `Fecha`. Dentro del cuerpo de la definición el miembro `dia` del objeto referenciado por `fechanueva` es asignado al miembro `dia` del objeto actual, el cual es repetido entonces para los miembros `mes` y anio. Asignaciones como `a.operator=(b);` pueden usarse entonces para llamar al operador de asignación sobrecargado y asignar los valores del miembro `b` a `a`. Por comodidad, la expresión `a.operator=(b)` puede reemplazarse con `a = b;`. El programa 10.2 contiene el nuevo `operator` de asignación dentro del contexto de un programa completo.

Excepto por la adición de la declaración y definición del operador de asignación sobrecargado, el programa 10.2 es idéntico al programa 10.1 y produce la misma salida. Su utilidad para nosotros es que ilustra cómo pueden construirse de manera explícita definiciones de asignación propias. Sin embargo, antes de continuar, necesario hacer dos modificaciones simples a nuestro operador de asignación.



Programa 10.2

```
#include <iostream>
#include <iomanip>
using namespace std;

// declaración de clase
class Fecha
{
  private:
    int mes;
    int dia;
    int anio;
  public:
    Fecha(int = 7, int = 4, int = 2005);    // constructor
    void operator=(Fecha&);   // define la asignación de una fecha
    void mostrarFecha();      // función miembro para desplegar una fecha
};

// sección de implementación
Fecha::Fecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;
}
```

(Continúa)

[1] Es preferible transmitir una referencia a transmitir un objeto por valor debido a que reduce la sobrecarga requerida para hacer una copia de los miembros de datos de cada objeto.

(Continuación)

```
void Fecha::operator=(Fecha& fechanueva)
{

  dia = fechanueva.dia;       // asigna el dia
  mes = fechanueva.mes;       // asigna el mes
  anio = fechanueva.anio;     // asigna el año
  return;
}

void Fecha::mostrarFecha()
{
  cout << setfill('0')
       << setw(2) << mes << '/'
       << setw(2) << dia << '/'
       << setw(2) << anio % 100;
  return;
}
```

```
int main()
{
  Fecha a(4,1,2007), b(12,18,2008); // declara dos objetos

  cout << "\nLa fecha almacenada en a originalmente es ";
  a.mostrarFecha(); // despliega la fecha original
  a = b;            // asigna el valor de b a a
  cout << "\nDespués de la asignación la fecha almacenada en a es ";
  a.mostrarFecha(); // despliega los valores de a
  cout << endl;

  return 0;
}
```

Primero, para evitar cualquier alteración inadvertida al objeto usado en el lado derecho de la asignación, deberá usarse un argumento de referencia constante. Para la clase `Fecha`, éste adopta la forma:

```
void Fecha::operator=(const Fecha& fechanueva);
```

La modificación final se refiere al valor que devuelve la operación. Como se construyó, el operador de asignación simple no devuelve ningún valor, lo cual evita que se use en asignaciones múltiples como `a = b = c`. La razón para esto es que los operadores sobrecargados retienen la misma precedencia y asociatividad que sus versiones integradas equivalentes. Por tanto, una expresión como `a = b = c` es evaluada en el orden `a = (b = c)`. Por desgracia, en la forma en que se ha definido la asignación, la expresión `b = c` no devuelve ningún valor, lo que hace que la asignación subsiguiente a `a` sea un error. Para proporcionar asignaciones múltiples, una operación de asignación más completa devolvería una referencia a su tipo de clase. Como la implementación de dicha asignación requiere un apuntador de clase especial, la presentación de este operador de asignación más completo se pospone hasta que se introduzca el material presentado en el capítulo 11. Hasta entonces, el operador de asignación simple será más adecuado para nuestras necesidades.

Constructores de copia

Aunque la asignación luce similar a la inicialización, vale la pena señalar que son dos operaciones diferentes por completo. En C++ ocurre una inicialización cada vez que se crea un objeto nuevo. En una asignación no se crea un objeto nuevo, tan sólo se cambia el valor de un objeto existente. La figura 10.1 ilustra esta diferencia.

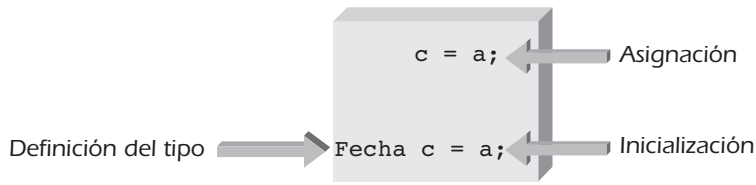


Figura 10.1 *Inicialización y asignación.*

Un tipo de inicialización que se parece mucho a la asignación ocurre en C++ cuando un objeto es inicializado usando otro objeto de la misma clase. Por ejemplo, en la declaración

```
Fecha b = a;
```

o su forma equivalente por completo

```
Fecha b(a);
```

El objeto `b` es inicializado con un objeto `a` declarado previamente. El constructor que ejecuta este tipo de inicialización se llama **constructor de copia** y si no declara uno el compilador elaborará uno para usted. El **constructor de copia por omisión** del compilador se ejecuta de una manera similar al `operator` de asignación por omisión al hacer una copia miembro por miembro entre objetos. Por tanto, para la declaración `Fecha b = a;` el constructor de copia por omisión establece los valores de mes, día y año de `b` con las contrapartes respectivas en `a`. Como con los operadores de asignación por omisión, los constructores de copia por omisión funcionan bien a menos que la clase contenga apuntadores a miembros de datos. Antes de considerar las complicaciones que pueden ocurrir con los apuntadores a miembros de datos y cómo manejarlos, será útil ver cómo construir nuestros propios constructores de copias.

Los constructores de copias, como todas las funciones de clase, son declarados en la sección de declaración de clase y se definen en la sección de implementación de clase. La declaración de un constructor de copia tiene la forma general:

```
nombreClase(const nombreClase&);
```

Como en todos los constructores, el nombre de la función debe ser el nombre de la clase. Como lo ilustra además la declaración, el argumento es una referencia a la clase, lo cual es una característica de todos los constructores de copia.[2] Para asegurar que el argumento no es alterado de manera inadvertida, siempre se especifica como una constante. Aplicando esta forma general a la clase `Fecha`, un constructor de copia puede declararse de manera explícita como

```
Fecha(const Fecha&);
```

[2]Un constructor de copia se define con frecuencia como un constructor cuyo primer argumento es una referencia a su tipo de clase, con los argumentos adicionales como valores por omisión.

La implementación real de este constructor, a fin de que ejecute la misma inicialización miembro por miembro que el constructor de copia por omisión, debería adoptar la forma:

```
Fecha::Fecha(const Fecha& fechavieja)
{
  mes = fechavieja.mes;
  dia = fechavieja.dia;
  anio = fechavieja.anio;
}
```

Como con el operador de asignación, el uso de un argumento de referencia para el constructor de copia no es accidental: el argumento de referencia facilita una vez más una notación simple dentro del cuerpo de la función. El programa 10.3 contiene este constructor de copia dentro del contexto de un programa completo.

Programa 10.3

```
#include <iostream>
#include <iomanip>
using namespace std;

// declaración de clase

class Fecha
{
  private:
    int mes;
    int dia;
    int anio;
  public:
    Fecha(int = 7, int = 4, int = 2005);   // constructor
    Fecha(const Fecha&);   // constructor de copia
    void mostrarFecha();   // función miembro para desplegar una fecha
};

// sección de implementación

Fecha::Fecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;
}
```

(Continúa)

(Continuación)

```
Fecha::Fecha(const Fecha& fechavieja)
{
  mes = fechavieja.mes;
  dia = fechavieja.dia;
  anio = fechavieja.anio;
}

void Fecha::mostrarFecha()
{
  cout << setfill('0')
       << setw(2) << mes << '/'
       << setw(2) << dia << '/'
       << setw(2) << anio % 100;

  return;
}
```

```
int main()
{
  Fecha a(4,1,2007), b(12,18,2008); // usa el constructor
  Fecha c(a);    // usa el constructor de copia
  Fecha d = b;   // usa el constructor de copia

  cout << "\nLa fecha almacenada en a es ";
  a.mostrarFecha();
  cout << "\nLa fecha almacenada en b es ";
  b.mostrarFecha();
  cout << "\nLa fecha almacenada en c es ";
  c.mostrarFecha();
  cout << "\nLa fecha almacenada en d es ";
  d.mostrarFecha();
  cout << endl;

  return 0;
}
```

La salida producida por el programa 10.3 es:

```
La fecha almacenada en a es 04/01/07
La fecha almacenada en b es 12/18/08
La fecha almacenada en c es 04/01/07
La fecha almacenada en d es 12/18/08
```

Como lo ilustra esta salida, los miembros de datos de `c` y `d` han sido inicializados por el constructor de copia con los valores de `a` y `b`, respectivamente. Aunque el constructor de copia definido en el programa 10.3 no añade nada a la funcionalidad proporcionada por el constructor de copia por omisión del compilador, proporciona los fundamentos para definir nuestros propios constructores de copia.

Inicialización base/miembro[3]

Excepto por los nombres de referencia `fechavieja` y `fechanueva`, una comparación del constructor de copia del programa 10.3 con el operador de asignación del programa 10.2 muestra que en esencia son la misma función. La diferencia en estas funciones es que el constructor de copia crea primero los miembros de datos del objeto antes que el cuerpo del constructor use la asignación para especificar valores de miembros. Por tanto, el constructor de copia no ejecuta una inicialización verdadera, sino más bien una creación seguida por asignación.

Una inicialización verdadera no dependería en absoluto de la asignación y es posible en C++ usando una **lista de inicialización base/miembro**. Dicha lista sólo puede aplicarse a funciones constructoras y puede escribirse de dos maneras.

La primera forma de construir una lista de inicialización base/miembro es dentro de la sección de declaración de una clase usando la forma

```
nombreClase(lista de argumentos) : lista de miembros de datos(valores de
inicialización) {}
```

Por ejemplo, usando esta forma, un constructor por omisión que ejecuta una inicialización verdadera es

```
// sección de declaración de clase

public:
  Fecha(int me = 4, int da = 1, int ao = 2006) : mes(me), dia(da), anio(ao) {}
```

La segunda forma es declarar un prototipo de función con valores por omisión en la sección de declaración de la clase seguido por la lista de inicialización en la sección de implementación. Para el constructor `Fecha` esto adopta la forma:

```
// sección de declaración de clase

public:
  Fecha(int = 4, int = 1, int = 2006); // prototipo con valores por
omisión

// sección de implementación de clase

Fecha::Fecha(int me, int da, int ao) : mes(me), dia(da), anio(ao) {}
```

Hay que observar que en ambas formas el cuerpo de la función constructora está vacío. Esto no es un requisito, y el cuerpo puede incluir cualesquiera operaciones subsiguientes que le gustaría que ejecutara el constructor. Lo interesante de este tipo de constructor es que diferencia con claridad entre las tareas de inicialización ejecutadas en la lista de inicialización de miembros contenida entre los dos puntos y las llaves, y cualesquiera asignaciones subsiguientes que pudieran estar contenidas dentro del cuerpo de la función. Aunque no se usará este tipo de inicialización en lo subsiguiente, se requiere siempre que haya una variable instanciada `const class`.

[3]El material en esta sección se presenta sólo como complemento y puede omitirse sin perder la continuidad del tema.

Ejercicios 10.1

1. Describa la diferencia entre asignación e inicialización.

2. **a.** Construya una clase llamada `Tiempo` que contenga tres miembros de datos en número entero llamados `horas, min` y `seg` que se usarán para almacenar las horas, minutos y segundos. Los miembros de la función deberán incluir un constructor que proporcione valores por omisión de 0 para cada miembro de datos, una función de despliegue que imprima los valores de datos de un objeto y un operador de asignación que ejecute una asignación miembro por miembro entre dos objetos de tiempo.
   b. Incluya la clase `Tiempo` elaborada en el ejercicio 2a en un programa funcional en C++ que cree y despliegue dos objetos `Tiempo`, al segundo de los cuales se le asignen los valores del primer objeto.

3. **a.** Construya una clase llamada `Complejo` que contenga dos miembros de datos de precisión doble llamados `real` e `imag`, las cuales se usarán para almacenar las partes real e imaginaria de un número complejo. Los miembros de la función deberán incluir un constructor que proporcione valores por omisión de 0 para cada función miembro, una función de despliegue que imprima los valores de datos de un objeto y un operador de asignación que ejecute una asignación miembro por miembro entre dos objetos de números complejos.
   b. Incluya la clase escrita para el ejercicio 3a en un programa funcional en C++ que cree y despliegue los valores de dos objetos `Complejo`, al segundo de los cuales se le asignen los valores del primer objeto.

4. **a.** Construya una clase llamada `Cartesiano` que contenga dos miembros de datos de precisión doble llamados `x` y `y`, los cuales se usarán para almacenar los valores `x` y `y` de un punto en coordenadas rectangulares. Los miembros de la función deberán incluir un constructor que inicialice los valores `x` y `y` de un objeto en 0 y funciones para entrada y despliegue de los valores `x` y `y` de un objeto. Además, deberá haber una función de asignación que ejecute una asignación miembro por miembro entre dos objetos `Cartesiano`.
   b. Incluya la clase escrita para el ejercicio 4a en un programa funcional en C++ que cree y despliegue los valores de dos objetos `Cartesiano`, al segundo de los cuales se le asignarán los valores del primer objeto.

5. **a.** Construya una clase llamada `Auto` que contenga los siguientes tres miembros de datos: una variable de precisión doble llamada `tamanioMotor`, una variable de carácter llamada `estiloModelo` y una variable en número entero llamada `codigoColor`. Las funciones de clase deberán incluir un constructor que proporcione valores por omisión de 0 para cada miembro de datos numérico y una `X` para cada variable de carácter; una función de despliegue que imprima el tamaño del motor, el estilo del modelo y el código de color; y un operador de asignación que ejecute una asignación miembro por miembro entre dos objetos `Auto` para cada variable de instancia.
   b. Incluya la clase escrita para el ejercicio 5a en un programa funcional en C++ que cree y despliegue dos objetos `Auto`, al segundo de los cuales se le asignan los valores del primer objeto.

Punto de información

Valores e identidades

Aparte de cualquier comportamiento que se le suministre a un objeto, una peculiaridad que comparten los objetos con las variables es que siempre tienen una identidad única. Es la identidad de un objeto la que permite distinguir un objeto de otro. Esto no sucede con un valor, como el número 5, porque todas las ocurrencias de 5 son indistinguibles entre sí. Como tales, los valores no se consideran como objetos en lenguajes de programación orientados a objetos como C++.

Otra característica distintiva entre un objeto y un valor es que un valor nunca puede ser un contenedor cuyo valor pueda cambiar, mientras que es claro que un objeto sí puede hacerlo. Un valor tan sólo es una entidad independiente.

Ahora considérese una cadena como "Chicago". Como una cadena, es un valor. Sin embargo, en vista que Chicago también podría ser un objeto específico e identificable del tipo `Ciudad`, es importante el contexto en que se usa el nombre. Hay que observar que si la cadena "Chicago" fuera asignada al atributo del nombre de un objeto, volvería a ser un valor.

10.2 Características adicionales de la clase

Esta sección presenta características adicionales relacionadas con las clases. Éstas incluyen el alcance de una clase, la creación de miembros estáticos de clase y la concesión de privilegios de acceso a las funciones que no son miembros. Cada uno de estos temas puede leerse de manera independiente de los demás.

Alcance de la clase

Ya se ha esbozado el alcance local y global en la sección 6.5. Como se vio, el alcance de una variable define la porción de un programa en la cual puede haber acceso a la variable.

Para variables locales, este alcance es definido por cualquier bloque contenido dentro de un par de llaves, `{}`. Esto incluye tanto el cuerpo de la función completo como los subbloques internos. Además, todos los parámetros de una función se consideran como variables locales de función.

Las variables globales son accesibles desde su punto de declaración a través de la porción restante del archivo que las contiene, con tres excepciones:

1. Si una variable local tiene el mismo nombre que una variable global, sólo puede haber acceso a la variable global dentro del alcance de la variable local usando el operador de resolución de alcance, `::`.
2. El alcance de una variable global puede extenderse a otro archivo usando la palabra clave `extern`.
3. El mismo nombre global puede reutilizarse en otro archivo para definir una variable separada y distinta usando la palabra clave `static`. Las variables globales `static` son desconocidas fuera de su archivo inmediato.

Además de los alcances local y global, cada clase también define un alcance de **clase asociado**. Es decir, los nombres de los miembros de datos y de función son locales al alcance de su clase. Por tanto, si el nombre de una variable global se reutiliza dentro de una clase, la variable local es ocultada por el miembro de datos de clase de la misma manera en que una variable de función local oculta a una variable global del mismo nombre. Asimismo, los nombres de la función miembro son locales para la clase en la que son declarados y sólo pueden ser usados por objetos declarados para la clase. Además, las variables de funciones locales también ocultan los nombres de los miembros de datos de clase que tienen el mismo nombre. La figura 10.2 ilustra el alcance de las variables y funciones para las siguientes declaraciones.

```
double tasa;    // global
// declaración de clase
class Prueba
{
  private:
    double cantidad, precio, total;    // alcance de clase
  public:
    double extend(double, double);    // alcance de clase
};
```

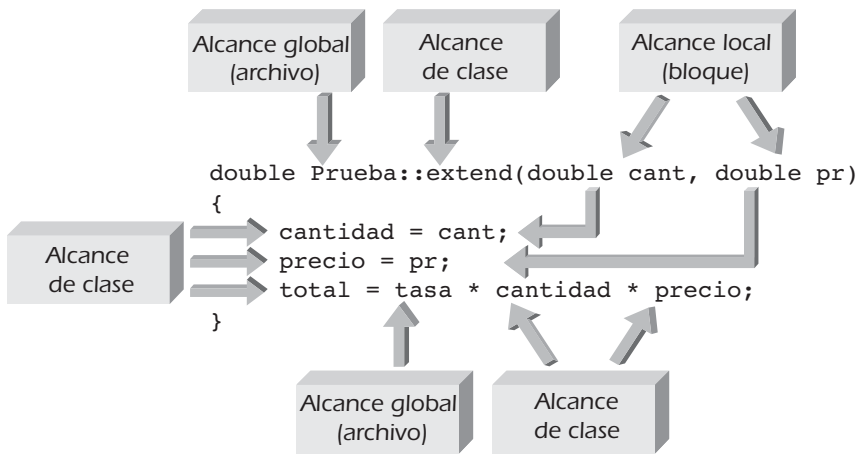


Figura 10.2 Ejemplo de alcances.

Miembros de clase tipo static

Conforme se crea cada objeto de clase obtiene su propio bloque de memoria para sus miembros de datos. En algunos casos, sin embargo, es conveniente para cada instanciación de una clase compartir la *misma* ubicación de memoria para una variable específica. Por ejemplo, considere una clase consistente en información de nómina de empleados, donde cada empleado es sujeto a la misma tasa de impuesto para seguridad social. Es claro que podríamos hacer el impuesto de ventas una variable global, pero esto no es muy seguro. Tales datos podrían modificarse en cualquier parte del programa, podrían estar en

conflicto con un nombre de variable idéntico dentro de una función y por supuesto violan el principio de ocultamiento de datos de C++.

Este tipo de situación se maneja en C++ declarando que una variable de clase es estática. Los miembros de datos de la clase `static` comparten el mismo espacio de almacenamiento para todos los objetos de la clase; como tales, actúan como variables globales para la clase y proporcionan un medio de comunicación entre objetos.

C++ requiere que dichas variables `static` sean declaradas dentro de la sección de declaración de la clase. Para crear estas variables, la variable `static` declarada debe ser redeclarada, con o sin un valor inicial (esto define la variable en contraste con una instrucción de declaración formal que no asigna físicamente almacenamiento para la variable) fuera de las secciones de declaración de la clase.

Por ejemplo, suponga la declaración de clase:

```
//declaración de clase

class Empleado
{
  private:
    static double tasa_imp;
    int num_id;
  public:
    Empleado(int); // constructor
    void desplegar();
};
```

La definición e inicialización de la variable `static tasa_imp` se logra usando una instrucción como la siguiente:

```
double Empleado::tasa_imp = 0.07; // esto define tasa_imp
```

Aquí se usa el operador de resolución de alcance, `::`, para identificar `tasa_imp` como miembro de la clase Empleado y no se incluye la palabra clave `static`. El programa 10.4 usa esta definición dentro del contexto de un programa completo.

La salida producida por el programa 10.4 es

```
El empleado número 11122 tiene una tasa de impuestos de
0.07

El empleado número 11133 tiene una tasa de impuestos de
0.07
```

Aunque podría parecer que la inicialización de `tasa_imp` es global, no lo es. Una vez que se hace la definición, cualquier otra definición producirá un error. Por tanto, la definición real de un miembro `static` sigue siendo responsabilidad del creador de la clase. El espacio de almacenamiento compartido producido por el miembro de datos `static` y los objetos creados en el programa 10.4 se ilustran en la figura 10.3.

Programa 10.4

```
#include <iostream>
using namespace std;

// declaración de clase

class Empleado
{
  private:
    static double tasa_imp;
    int num_id;
  public:
    Empleado(int = 0);   // constructor
    void desplegar();    // función de acceso
};

// definición del miembro static
double Empleado::tasa_imp = 0.07; // esto define tasa_imp

// sección de implementación

Empleado::Empleado(int num)
{
  num_id = num;
}

void Empleado::desplegar()
{
  cout << "El empleado número " << num_id
       << " tiene una tasa de impuestos de " << tasa_imp << endl;

  return;
}

int main()
{
  Empleado emp1(11122), emp2(11133);

  emp1.desplegar();
  emp2.desplegar();

  return 0;
}
```

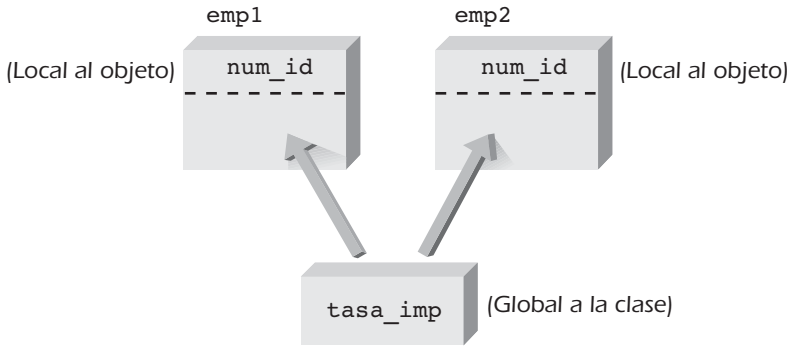


Figura 10.3 *Compartiendo el miembro de datos* `static` `tasa_imp`.

Además de los miembros de datos `static`, también pueden crearse funciones miembro `static`. Dichas funciones se aplican a los la clase en conjunto en lugar de a los objetos de clase individuales y sólo pueden tener acceso a los miembros de datos `static` y otras funciones miembro `static` de la clase. En el programa 10.5 se proporciona un ejemplo de una función así.

Programa 10.5

```
#include <iostream>
using namespace std;

// declaración de clase

class Empleado
{
  private:
    static double tasa_imp;
    int num_id;
  public:
    Empleado(int = 0);   // constructor
    void desplegar();    // función de acceso
    static void desp();  // función static
};
// definición del miembro static
double Empleado::tasa_imp = 0.07;
```

(Continúa)

(Continuación)

```
// sección de implementación

Empleado::Empleado(int num)
{
  num_id = num;
}

void Empleado::desplegar()
{
  cout << "El empleado número " << num_id
       << " tiene una tasa de impuestos de " << tasa_imp << endl;

  return;
}

void Empleado::desp()
{
  cout << "La tasa de impuestos estática es " << tasa_imp << endl;

  return;
}
```

```
int main()
{
  Empleado::desp();   // llama a las funciones static
  Empleado emp1(11122), emp2(11133);

  emp1.desplegar();
  emp2.desplegar();

  return 0;
}
```

La salida producida por el programa 10.5 es

```
La tasa de impuestos estática es 0.07
El empleado número 11122 tiene una tasa de impuestos de 0.07
El empleado número 11133 tiene una tasa de impuestos de 0.07
```

Una vez que se hace la definición de una variable de clase `static` (en oposición a la declaración), cualquier otra definición producirá un error. Por tanto, la definición real de un miembro `static` sigue siendo responsabilidad del creador de la clase. Si se omite esta definición ocurrirá un error de compilador.

Funciones amigas

El único método que existe en la actualidad para tener acceso y manipular a variables private de una clase es por medio de las funciones miembro de la clase. Desde el punto de vista conceptual, este arreglo puede presentarse como se ilustra en la figura 10.4a. Sin embargo, hay ocasiones en que es útil proporcionar dicho acceso a funciones selectas que no son miembros.

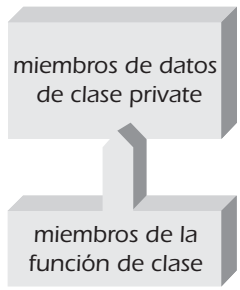


Figura 10.4a *Se proporciona acceso directo a funciones miembro.*

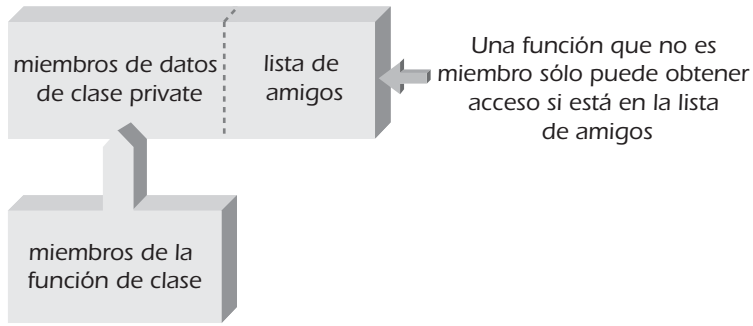


Figura 10.4b *Acceso proporcionado a funciones que no son miembros.*

El procedimiento para proporcionar este acceso externo es bastante simple: la clase mantiene su propia lista aprobada de funciones no miembros a las que se conceden los mismos privilegios que a las funciones de una clase. Las funciones no miembros en la lista se llaman ***funciones amigas*** y la lista se conoce como ***lista de amigos***.

La figura 10.4b ilustra conceptualmente el uso de dicha lista para el acceso a no miembros. Cualquier función que intente tener acceso a las variables privadas de un objeto es comprobada primero contra la lista de amigos: si la función está en la lista, se aprueba el acceso; de lo contrario se niega el acceso.

Desde un punto de vista de codificación la lista de amigos tan sólo es una serie de declaraciones de prototipo de función que son precedidas por la palabra clave `friend` y se incluyen en la sección de declaración de la clase. Por ejemplo, si a las funciones nombradas `agreg_real()` y `agreg_imag()` se les va a permitir el acceso a los miembros privados de una clase llamada `Complejo`, se incluirían los siguientes prototipos dentro de la sección de declaración de `Complejo`.

```
friend double agreg_real(Complejo&, Complejo&);
friend double agreg_imag(Complejo&, Complejo&);
```

Aquí la lista de amigos consiste de dos declaraciones. Los prototipos indican que cada función devuelve un número en punto flotante y espera dos referencias a objetos del tipo `Complejo` como argumentos. El programa 10.6 incluye estas dos declaraciones amigas en un programa completo.

Programa 10.6

```
#include <iostream>
#include <cmath>
using namespace std;

// declaración de clase

class Complejo
{
  // lista de amigos
  friend double agreg_real(Complejo&, Complejo&);
  friend double agreg_imag(Complejo&, Complejo&);
  private:
    double real;
    double imag;
  public:
    Complejo(double = 0, double = 0); // constructor
    void desplegar();

};

// sección de implementación

Complejo::Complejo(double rl, double im)
{
  real = rl;
  imag = im;
}

void Complejo::desplegar()
{
  char signo = '+';

  if(imag < 0) signo = '-';
  cout << real << signo << abs(imag) << 'i';

  return;
}
```

(Continúa)

(Continuación)

```
// implementación de la amistad

double agreg_real(Complejo &a, Complejo &b)
{
  return(a.real + b.real);
}

double agreg_imag(Complejo &a, Complejo &b)
{
  return(a.imag + b.imag);
}

int main()
{
  Complejo a(3.2, 5.6), b(1.1, -8.4);
  double re, im;

  cout << "\nEl primer número complejo es ";
  a.desplegar();
  cout << "\nEl segundo número complejo es ";
  b.desplegar();

  re = agreg_real(a,b);
  im = agreg_imag(a,b);
  Complejo c(re,im); // crea un nuevo objeto Complejo
  cout << "\n\nLa suma de estos dos números complejos es ";
  c.desplegar();

  return 0;
}
```

La salida producida por el programa 10.6 es

```
El primer número complejo es 3.2+5.6i
El segundo número complejo es 1.1-8.4i

La suma de estos dos números complejos es 4.3-2.8i
```

Al revisar el programa 10.6 hay que observar cuatro elementos. El primero es que en vista que los amigos no son miembros de la clase, no son afectados por la sección de acceso en que son declarados; *pueden ser declarados en cualquier parte dentro de la sección de declaración*. La convención que se ha seguido es incluir todas las declaraciones `friend` inmediatamente después del encabezado de la clase. El segundo elemento que hay que observar es que la palabra clave `friend` (como la palabra clave `static`) se usa sólo dentro de la declaración de la clase y no en la definición de función real. Tercero, dado que se pretende que una función `friend` tenga acceso a las variables privadas de un objeto, al menos uno de los argumentos del amigo deberá ser una referencia a un objeto de la clase que lo ha hecho un amigo. Por último, como se ilustra en el programa 10.6, es la clase la que concede la condición de `friend`

a una función y no al contrario. La función nunca puede conferirse condición de `friend` por sí misma, debido a que al hacerlo violaría los conceptos de ocultamiento de datos y acceso proporcionados por una clase.

Ejercicios 10.2

1. **a.** Vuelva a escribir el programa 10.5 para incluir una variable `static` en número entero llamada `num_emps`. Esta variable deberá actuar como un contador que se inicialice en cero y sea incrementado por el constructor de clase cada vez que se declare un objeto nuevo. Vuelva a escribir la función `static desp()` para desplegar el valor de este contador.
   b. Pruebe el programa escrito para el ejercicio 1a. Haga que la función `main()` llame a `desp()` después que se cree cada objeto Empleado.

2. **a.** Construya una clase llamada `Circulo` que contenga dos variables en número entero llamadas `centro_x` y `centro_y`, y una variable de precisión doble llamada radio. Además, la clase deberá contener un miembro de datos `static` llamado `factor_escala`. Aquí los `valores centro_x` y `centro_y` representan el punto central de un círculo, radio representa el radio real del círculo y `factor_escala` representa un factor de escala que se usará para cambiar la escala del círculo para que se ajuste a una variedad de dispositivos de despliegue.
   b. Incluya la clase escrita para el ejercicio 2a en un programa funcional en C++.

3. **a.** Las siguientes tres instrucciones en el programa 10.6

   ```
   re = agreg_real(a,b);
   im = agreg_imag(a,b);
   Complejo c(re,im);  // crea un objeto Complejo nuevo
   ```

   podrían reemplazarse por la instrucción simple

   ```
   Complejo c(agreg_real(a,b), agreg_imag(a,b));
   ```

   b. Verifique su respuesta al ejercicio 3a ejecutando el programa 10.6 reemplazando la instrucción sugerida.

4. **a.** Vuelva a escribir el programa escrito para el ejercicio 2a, pero incluya una función `friend` que multiplique el radio de un objeto por un `factor_escala static` y luego despliegue el valor del radio real y el valor a escala.
   b. Pruebe el programa escrito para el ejercicio 4a.

5. Vuelva a escribir el programa 10.6 para que sólo tenga una función `friend` llamada `agregComplejo()`. Esta función deberá aceptar dos objetos complejos y devolver un objeto complejo. Las partes real e imaginaria del objeto devuelto deberán ser la suma de las partes real e imaginaria, respectivamente, de los dos objetos transmitidos a `agregComplejo()`.

6. a. Construya una clase llamada `Coord` que contenga dos variables de precisión doble llamadas `val_x` y `val_y`, las cuales se usarán para almacenar los valores `x` y `y` de un punto en coordenadas rectangulares. Las funciones de clase deberán incluir funciones constructoras y de despliegue apropiadas y una función `friend` llamada `conv_pol()`. La función `conv_pol()` deberá aceptar dos números en punto flotante que representen un punto en coordenadas polares y las convierta en coordenadas rectangulares. Para conversión de coordenadas polares a rectangulares use las fórmulas

$$x = r \cos \theta$$
$$y = r \sin \theta$$

b. Incluya la clase escrita para el ejercicio 6a en un programa funcional en C++.

7. a. Construya dos clases llamadas `coord_rec` y `coord_pol`. La clase llamada `coord_rec` deberá contener dos variables de precisión doble llamadas `val_x` y `val_y`, las cuales se usarán para almacenar los valores de `x` y `y` de un punto en coordenadas rectangulares. Las funciones de clase deberán incluir funciones constructoras y de despliegue apropiadas y una función `friend` llamada `conv_pol()`.

La clase llamada `coord_pol` deberá contener dos miembros de datos de precisión doble llamados `dist` y `theta`, los cuales se usarán para almacenar los valores de distancia y ángulo de un punto representado en coordenadas polares. Las funciones de clase deberán incluir funciones constructoras y de despliegue apropiadas y una función `friend` llamada `conv_pol()`. La función `friend` deberá aceptar un argumento en número entero llamado `dir`, dos argumentos de precisión doble llamados `val1` y `val2`, y dos argumentos de referencia llamados `ref_rec` y `ref_pol`, el primero de los cuales deberá ser una referencia a un objeto del tipo `coord_rec`, y el segundo a un objeto del tipo `coord_pol`. Si el valor de `dir` es `1`, `val1` y `val2` se considerarán como coordenadas rectangulares `x` y `y` que se van a convertir en coordenadas polares; si el valor de `dir` es cualquier otro valor, `val1` y `val2` se considerarán como valores de distancia y ángulo que se van a convertir en coordenadas polares. Para la conversión de rectangular a polar, las coordenadas son

$$r = \sqrt{x^2 + y^2}$$
$$\theta = \tan (y/x)^{-1}$$

Para la conversión de coordenadas polares a rectangulares, use las fórmulas:

$$x = r \cos \theta$$
$$y = r \sin \theta$$

b. Incluya la clase escrita para el ejercicio 7a en un programa funcional en C++.

10.3 FUNCIONES OPERADOR

En la sección 10.1 se construyó un operador de asignación simple. En esta sección se extenderá esta capacidad y se mostrará cómo ampliar los operadores integrados de C++ para trabajar con objetos de clase. Como se descubrirá, los operadores de clase son en sí mismos funciones miembros o amigas.

Los únicos símbolos permitidos para propósitos definidos por el usuario son el subconjunto de símbolos integrados de C++ mostrados en la tabla 10.1. Cada uno de estos símbolos puede adoptarse para uso de la clase sin limitación en cuanto a su significado.[4] Esto se logra al hacer que cada operación sea una función que puede ser sobrecargada como cualquier otra función.

La operación de los símbolos mostrados en la tabla 10.1 puede redefinirse para que se adapten a nuestras clases, sujeta a las siguientes restricciones:

- Los símbolos que no están en la tabla 10.1 no pueden redefinirse. Por ejemplo, los símbolos `.`, `::` y `?:` no pueden redefinirse.
- No pueden crearse símbolos de operadores nuevos. Por ejemplo, en vista que `%%` no es un operador en C++, no puede definirse como un operador de clase.
- No pueden modificarse la precedencia ni la asociatividad de los operadores de C++. Por tanto, no puede darse al operador de adición una precedencia mayor que al operador de multiplicación.
- No pueden redefinirse los operadores para los tipos integrados de C++.
- Un operador de C++ que es unario no puede cambiarse a un operador binario y un operador binario no puede cambiarse a un operador unario.
- El operador debe ser un miembro de una clase o definirse para adoptar al menos un miembro de la clase como operando.

El primer paso en la tabla 10.1 para proporcionar operadores a una clase es decidir cuáles operaciones tienen sentido para la clase y cómo deberían definirse. Como un ejemplo específico, se continuará construyendo la clase `Fecha` introducida con anterioridad. Para esta clase se define un conjunto significativo pequeño de operaciones de clase.

Es claro que la adición de dos fechas no es significativa. Sin embargo, la adición de una fecha con un número entero tiene sentido si el número entero se toma como el número de días que se van a sumar a la fecha. Del mismo modo, la sustracción de un número entero de una fecha tiene sentido. Además, la sustracción de dos fechas es significativa si se define la diferencia como el número de días entre las dos fechas. Asimismo, tiene sentido comparar dos fechas y determinar si las fechas son iguales o si una fecha ocurre antes o después que otra fecha. Ahora se verá cómo pueden implementarse estas operaciones usando los símbolos de operador de C++.

[4] La única limitación es que la sintaxis del operador no puede cambiarse. Por tanto, un operador binario debe permanecer como tal y un operador unario debe permanecer así. Dentro de esta restricción de sintaxis un símbolo de operador puede utilizarse para producir cualquier operación, ya sea que la operación sea consistente con el uso aceptado del símbolo o no. Por ejemplo, podría redefinirse el símbolo de adición para proporcionar multiplicación. Es evidente que esto viola la intención y espíritu de poner estos símbolos a nuestra disposición. Hay que tener mucho cuidado en redefinir cada símbolo de una manera consistente con su uso aceptado.

Tabla 10.1 Operadores disponibles para uso de la clase

Operador	Descripción

`()`
`[]`
`->`
`new`
`delete`
	Llamada a una función
	Elemento de una matriz
	Referencia de apuntador a un miembro de una estructura
	Memoria asignada dinámicamente
	Memoria desasignada dinámicamente

`++`
`--`
`-`
`!`
`~`
`*`
	Incremento
	Decremento
	Menos unario
	Negación lógica
	Complemento a uno
	Indirección

`*`
`/`
`%`
`+`
`-`
	Multiplicación
	División
	Módulo (residuo)
	Adición
	Sustracción

`<<` `>>`	Cambio a la izquierda Cambio a la derecha

`<`
`<=`
`>`
`>=`
`==`
`!=`
`&&`
`||`
	Menor que
	Menor que o igual a
	Mayor que
	Mayor que o igual a
	Igual a
	Diferente a
	AND lógico
	OR lógico

`&`
`^`
`|`
	AND bit por bit
	OR exclusivo bit por bit
	OR inclusivo bit por bit

`=`
`+=  -=  *=`
`/=  %=  &=`
`^=  |=`
`<<=  >>=`
	Asignación
	Asignación
	Asignación
	Asignación
	Asignación

`,`	Coma

Una operación definida por el usuario se crea como una función que redefine los símbolos de operador integrados de C++ para uso de la clase. Las funciones que definen las operaciones en objetos de clase y usan símbolos de operador integrados de C++ se conocen como **funciones operador**.

Las funciones operador se declaran e implementan de la misma manera que todas las funciones miembro, con una excepción: es el nombre de la función el que conecta al símbolo operador apropiado con la operación definida por la función. El nombre de una función operador siempre tiene la forma operador<símbolo> donde <símbolo> es uno de los operadores mostrados en la tabla 10.1. Por ejemplo, el nombre de la función `operator+` es el nombre de la función de adición, mientras el nombre de la `función operator==` es el nombre de la función de comparación igual a.

Una vez que se ha seleccionado el nombre de función apropiado, el proceso de escribir la función tan sólo equivale a hacer que se acepten las entradas deseadas y se produzca el valor devuelto correcto.[5] Por ejemplo, al comparar dos objetos `Fecha` en cuanto a igualdad se seleccionaría el operador de igualdad de C++. Por tanto, el nombre de la función se vuelve `operator==`. Podría desearse que la operación de comparación acepte dos objetos `Fecha`, los compare en forma interna y devuelva un valor en número entero indicando el resultado de la comparación: `verdadero` para igualdad y `falso` para desigualdad. Como una función miembro, un prototipo adecuado que podría incluirse en la sección de declaración de la clase es

```
bool operator==(Fecha&);
```

Este prototipo indica que la función se llama `operator==`, que devuelve un valor booleano y que acepta una referencia a un objeto Fecha.[6] Sólo se requiere aquí un objeto `Fecha` porque el segundo objeto `Fecha` será el objeto que llame a la función. Ahora escribiremos la definición de la función que se incluirá en la sección de implementación de la clase. Suponiendo que la clase se llama Fecha, una definición adecuada es

```
bool Fecha::operator==(Fecha& Fecha2)
{
  if( dia == fecha2.dia && mes == fecha2.mes && anio == fecha2.anio)
    return verdadero;
  else
    return falso;
}
```

Una vez que se ha definido esta función, puede invocarse usando la misma sintaxis que para los tipos integrados de C++. Por ejemplo, si `a` y `b` son objetos del tipo `Fecha`, la expresión `if (a == b)` es válida. El programa 10.7 incluye esta instrucción if al igual que la declaración y definición de esta función operador dentro del contexto de un programa completo.

La salida producida por el programa 10.7 es

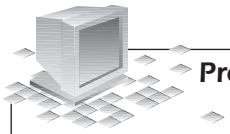
```
Las fechas a y b no son iguales.
Las fechas a y c son iguales.
```

La primera característica nueva que se ilustra en el programa 10.7 es la declaración e implementación de la función llamada `operator==()`. Excepto por su nombre, esta función operador se construye de la misma manera que cualquier otra función miembro: se declara en la sección de declaración y se define en la sección de implementación. La segunda característica nueva es cómo se invoca la función. Las funciones operador pueden invocarse usando sus símbolos asociados en lugar de la forma en que son llamadas otras funciones. Sin embargo, en vista que las funciones operador son funciones verdaderas, también puede utilizarse el método tradicional para llamarlas, especificando su nombre e incluyendo argumentos apropiados. Por tanto, además de ser llamadas por la expresión `a == b` en el programa 10.7, también podría haberse usado la llamada `a.operator==(b)`.

---

[5] Como se señaló antes, esto implica que el operador especificado puede redefinirse para ejecutar cualquier operación. La buena práctica de programación, sin embargo, se opone a tales redefiniciones.

[6] El prototipo `bool operator==(Fecha)` también funciona. Sin embargo, es preferible transmitir una referencia a transmitir un objeto debido a que reduce la sobrecarga de la llamada a la función. Esto se debe a que transmitir un objeto significa que debe hacerse una copia del objeto para la función llamada, mientras que transmitir una referencia le da a la función acceso directo al objeto cuya dirección es transmitida.



Programa 10.7

```
#include <iostream>
using namespace std;

// declaración de clase
class Fecha
{
  private:
    int mes;
    int dia;
    int anio;
  public:
    Fecha(int = 7, int = 4, int = 2005); // constructor
    bool operator==(Fecha &); // declara la función operator==
};

// sección de implementación
Date::Date(int mm, int dd, int yyyy)
{
  mes = mm;
  dia = dd;
  anio = aaaa;
}

bool Fecha::operator==(Fecha &fecha2)
{
  if(dia == fecha2.dia && mes == fecha2,mes && anio == fecha2.anio)
    return verdadero;
  else
    return false;
}

int main()
{
  Fecha a(4,1,2007), b(12.18.2008), c(4,1,2007); // declara 3 objetos

  if (a == b)
    cout << "Las fechas a y b son iguales." << endl;
  else
    cout << "Las fechas a y b no son iguales." << endl;

  if (a == c)
    cout << "Las fechas a y c son iguales." << endl;
  else
    cout << "Las fechas a y c no son iguales." << endl;

  return 0;
}
```

Ahora crearemos otro operador para la clase `Fecha`, un operador de adición. Como antes, crear este operador requiere que se especifiquen tres elementos:

1. El nombre de la función operador
2. El procesamiento que la función va a ejecutar
3. El tipo de datos, si hay alguno, que la función va a devolver

Es evidente que para la adición se usará la función operador llamada `operator+`. Habiendo seleccionado el nombre de la función ahora debe determinarse qué se desea que haga esta función, de manera específica en lo que se relaciona con los objetos `Fecha`. Como se señaló antes, la suma de dos fechas no tiene sentido. Sin embargo, sumar un número entero a una fecha es significativo cuando el entero representa el número de días antes o después de una fecha determinada. Aquí la suma de un número entero a un objeto `Fecha` tan sólo es otro objeto `Fecha`, el cual deberá ser devuelto por la operación de adición. Por tanto, un prototipo adecuado para la función de adición es

```
Fecha operator+(int);
```

Este prototipo se incluiría en la sección de declaración de la clase. Especifica que un número entero se va a sumar a un objeto de la clase y la operación devuelve un objeto Fecha. Por tanto, si a es un objeto `Fecha`, la llamada a la función `a.operator+(284)`, o su alternativa más usada, `a + 284`, deberá hacer que el número `284` se sume en forma correcta al valor de la fecha de `a`. Ahora debemos construir la función para lograr esto.

Construir la función requiere que primero seleccionemos una convención de fecha específica. Por simplicidad se adoptará la convención de fecha financiera que considera que cada mes consta de 30 días y cada año consta de 360 días. Utilizando esta convención, la función sumará primero el número entero de días al valor del día del objeto `Fecha` y luego ajustará el valor del día resultante para que caiga dentro del rango de 1 a 30 y el valor del mes para que caiga dentro del rango de 1 a 12. Una función que logra esto es

```
Fecha Fecha::operator+(int dias)
{
  Fecha temp; // una Fecha temporal para almacenar el
                 resultado

  temp.dia = dia + dias; // suma los días
  temp.mes = mes;
  temp.anio = anio;
  while (temp.día > 30) // ahora ajusta los meses
  {
    temp.mes++;
    temp.dia -= 30;
  }
  while (temp.mes > 12) // ajusta los años
  {
    temp.anio++;
    temp.mes -= 12;
  }
  return temp;     // se devuelven los valores en temp
}
```

Aquí la característica importante es el uso del objeto `temp`. El propósito de este objeto es asegurar que no se altere ninguno de los argumentos de la función, los cuales se vuelven los operandos del operador. Para entender esto considérese una instrucción como `b = a + 284;` que usa esta función operador, donde `a` y `b` son objetos de `Fecha`. Esta instrucción nunca debería modificar el valor de `a`. Mejor dicho, la expresión `a + 284` deberá producir un valor de `Fecha` que luego se asigna a `b`. El resultado de la expresión es, por supuesto, el objeto `Fecha temp` devuelto por la función `operator+()`. El programa 10.8 usa esta función dentro del contexto de un programa completo.

Programa 10.8

```
#include <iostream>
#include <iomanip>
using namespace std;

// declaración de clase

class Fecha
{
  private:
    int mes;
    int dia;
    int anio;
  public:
    Fecha(int = 7, int = 4, int = 2005);    // constructor
    Fecha operator+(int);    // sobrecarga el operador +
    void mostrarFecha();     // función miembro para desplegar una fecha
};
// sección de implementación
Fecha::Fecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;
}

Fecha Fecha::operator+(int dias)
{
  Fecha temp; // una fecha temporal para almacenar el resultado

  temp.dia = dia + dias; // suma los días
  temp.mes = mes;
  temp.anio = anio;
  while (temp.día > 30) // ahora ajusta los meses
```

(Continúa)

(Continuación)

```
  {
    temp.mes++;
    temp.dia -= 30;
  }
  while (temp.mes > 12) // ajusta los años
  {
    temp.anio++;
    temp.mes -= 12;
  }
  return temp;    // se devuelven los valores en temp
}
void Fecha::mostrarFecha()
{
  cout << setfill('0')
       << setw(2) << mes << '/'
       << setw(2) << dia << '/'
       << setw(2) << anio % 100
  return;
}

int main()
{
  Fecha a(4,1,2007), b; // declara dos objetos

  cout << "La fecha inicial es ";
  a.mostrarFecha();
  b = a + 284;    // suma 284 dias = 9 meses y 14 dias
  cout << "La nueva fecha es ";
  b.mostrarFecha();

  return 0;
}
```

La salida producida por el programa 10.8 es

```
La fecha inicial es 04/01/07
La fecha nueva es 01/15/08
```

Funciones operador como amigas

Las funciones operador en los programas 10.7 y 10.8 se han construido como miembros de clase. Una característica interesante de las funciones operador es que, excepto para las funciones operador `=`, `()`, `[]` y `->`, también pueden escribirse como funciones amigas. Por ejemplo, si la función `operator+()` usada en el programa 10.8 se escribiera como amiga, un prototipo adecuado de la sección de declaración es

```
friend Fecha operator+(Fecha& , int);
```

Hay que observar que la versión amiga contiene una referencia a un objeto `Fecha` que no está contenida en la versión de la función miembro. En todos los casos la versión amiga equivalente de una función operador miembro debe contener una referencia adicional a la clase que no es requerida por la función miembro.[7] Esta equivalencia se muestra en la tabla 10.2 para operadores unarios y binarios.

Tabla 10.2 Requerimientos para el argumento de la función operador

	Función miembro	Función amiga
Operador unario	1 implícito	1 explícito
Operador binario	1 implícito y 1 explícito	2 explícitos

La función `operator+()` del programa 10.8, escrita como una función amiga, es

```
Fecha operator+(Fecha& op1, int dias)
{
  Fecha temp; // una Fecha temporal para almacenar
                el resultado

  temp.dia = op1.dia + dias; // suma los días
  temp.mes = op1.mes;
  temp.anio = op1.anio;
  while (temp.dia > 30)    // ahora ajusta los meses
  {
    temp.mes++;
    temp.dia -= 30;
  }
  while (temp.mes > 12)   // ajusta los años
  {
    temp.anio++;
    temp.mes -= 12;
  }
  return temp;    // devuelve los valores en temp
}
```

La única diferencia entre esta versión y la versión miembro es el uso explícito de un argumento `Fecha` llamado `op1` (la elección de este nombre es por completo arbitraria) en la versión amiga. Esto significa que dentro del cuerpo de la función amiga las primeras tres instrucciones de asignación hacen referencia de manera explícita a los miembros de datos de `op1` como `op1.dia, op1.mes` y `op1.anio`, mientras la función miembro tan sólo se refiere a estos argumentos como `dia, mes` y `anio`.

Al hacer la determinación de sobrecargar un operador binario ya sea como una función operador amiga o miembro, puede aplicarse la siguiente convención: *las funciones amigas son más apropiadas para funciones binarias que no modifican ninguno de sus operandos, como ==, +, -, etc., mientras las funciones miembro son más apropiadas para funciones binarias, como =, +=, -=, etc., que se usan para modificar uno de sus operandos.*

[7] Este argumento extra es necesario para identificar el objeto correcto. Este argumento no es necesario cuando se usa una función miembro porque la función miembro "sabe" en cuál objeto está operando. El mecanismo de este "conocimiento" es suministrado por un argumento de la función miembro implícito llamado `this`.

Ejercicios 10.3

1. **a.** Defina una *función operador* relacional mayor que llamada `operator>()` que pueda usarse con la clase `Fecha` declarada en el programa 10.7.
   b. Defina una *función operador* menor que llamada `operator<()` que pueda usarse con la clase `Fecha` declarada en el programa 10.7.
   c. Incluya las funciones operador escritas para los ejercicios 1a y 1b en un programa funcional en C++.

2. **a.** Defina una función operador de sustracción llamada `operator-()` que pueda usarse con la clase `Fecha` definida en el programa 10.7. La sustracción deberá aceptar un argumento en número entero largo que represente el número de días que se van a restar de un objeto de fecha y devolver una `Fecha`. Al hacer la sustracción use la suposición que todos los meses constan de 30 días y todos los años de 360 días. Además, deberá hacerse un ajuste de fin de mes, si es necesario, que convierta cualquier día resultante de 31 a un día de 30, excepto si el mes es febrero. Si el mes resultante es febrero y el día es 29, 30 o 31, deberá cambiarse a 28.
   b. Defina otra función operador de sustracción llamada `operator-()` que pueda usarse con la clase `Fecha` definida en el programa 10.7. La sustracción deberá producir un número entero largo que represente la diferencia en días entre dos fechas. Al calcular la diferencia de días use la cuenta de días financiera que supone que todos los meses tienen 30 días y todos los años tienen 360 días.
   c. Incluya los operadores sobrecargados escritos para los ejercicios 2a y 2b en un programa funcional en C++.

3. **a.** Determine si la siguiente función operador de adición proporciona el mismo resultado que la función usada en el programa 10.8.

```
Fecha Fecha::operator+(int dias)   // devuelve un objeto Fecha
{
  Fecha temp;

  temp.dia = dia + dias;   // suma los días
  temp.mes = mes + int(dia/30);   // determina el total de meses
  temp.dia = temp.dia % 30;   // determina el día presente
  temp.anio = anio + int(temp.mes/12);   // determina el total de
                                            años
  temp.mes = temp.mes % 12;   // determina el mes presente

  return temp;
}
```

   b. Verifique su respuesta al ejercicio 3a incluyendo la función en un programa funcional en C++.

4. **a.** Vuelva a escribir la función operador relacional de igualdad en el programa 10.7 como una función amiga.
   b. Verifique la operación de la función operador amiga escrita para el ejercicio 4a incluyéndola dentro de un programa funcional en C++.

5. **a.** Construya un operador de adición para la clase `Complejo` declarada en el programa 10.6. Éste deberá ser una función miembro que sume dos números complejos y devuelva un número complejo.
   b. Agregue una función miembro de operador de multiplicación a la clase usada en el ejercicio 5a que multiplique dos números complejos y devuelva un número complejo.
   c. Verifique la operación de las funciones operador escritas para los ejercicios 5a y 5b incluyéndolas dentro de un programa funcional en C++.

6. **a.** Vuelva a escribir la función operador de adición en el programa 10.8 para que dé cuenta de los días reales en un mes, ignorando los años bisiestos. (*NOTA*: Esto requiere un arreglo para almacenar los días en cada mes.)
   b. Verifique la operación de la función operador escrita para el ejercicio 6a incluyéndola dentro de un programa funcional en C++.

10.4 CONVERSIONES DEL TIPO DE DATOS

La conversión de un tipo de datos integrado a otro se describió en la sección 3.2. Con la introducción de los tipos de datos definidos por el usuario, las posibilidades para la conversión entre tipos de datos se expanden a los siguientes casos:

- Conversión de un tipo integrado a un tipo integrado
- Conversión de un tipo integrado a un tipo de clase
- Conversión de un tipo de clase a un tipo integrado
- Conversión de un tipo de clase a un tipo de clase

La primera conversión es manejada ya sea por las reglas de conversión implícitas integradas de C++ o por su operador explícito de molde. El segundo tipo de conversión se hace usando un **constructor de conversión de tipo**. El tercero y cuarto tipos de conversión se hacen usando una **función operador de conversión**. En esta sección se presenta el medio específico de ejecutar cada una de estas conversiones.

Conversión de integrado a integrado

La conversión de un tipo de datos integrado a otro ya se ha presentado en las secciones 3.1 y 3.3. Para repasar este caso brevemente, este tipo de conversión es implícita o explícita.

Ocurre una conversión implícita en el contexto de una de las operaciones de C++. Por ejemplo, cuando se asigna un valor en punto flotante a una variable en número entero, sólo se almacena la porción entera del valor. La conversión está implícita en la operación y es ejecutada de manera automática por el compilador.

Ocurre una conversión explícita siempre que se usa un molde. En C++ existen dos notaciones de molde. Usando la notación antigua de C, un molde tiene la forma *(tipo-de-datos) expresión* mientras la notación más reciente de C++ tiene la forma tipo función *tipo-de-datos(expresión)*. Por ejemplo, las expresiones `(int)24.32` e `int(24.32)` causan que el valor de precisión doble 24.32 sea truncado al valor entero 24.

Conversión de integrado a clase

Los moldes definidos por el usuario para convertir un tipo de datos integrado a uno de clase se crean usando funciones constructoras. Un constructor cuyo primer argumento no es un miembro de su `class` y cuyos argumentos restantes, si hay alguno, tienen valores por omisión es un **constructor de conversión de tipo**. Si el primer argumento de un constructor de conversión de tipo es un tipo de datos integrado, el constructor puede usarse para moldear el tipo de datos integrado en un objeto de clase. Es evidente que una restricción de tales funciones es que, como constructoras, deber ser funciones miembro.

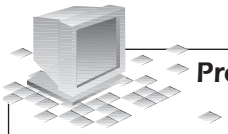
Aunque este tipo de molde ocurre cuando el constructor es invocado para inicializar un objeto, en realidad es un molde más general que podría ser evidente a primera vista. Esto se debe a que una función constructora puede ser invocada en forma explícita después que todos los objetos han sido declarados, se haya invocado antes o no como parte de una declaración del objeto. Antes de explorar más esto, se construirá primero un constructor de conversión de tipo. Luego se verá cómo usarlo como un molde independiente de su propósito de inicialización.

El molde que se construirá convertirá un número entero largo en un objeto `Fecha`. El objeto `Fecha` consistirá en fechas en la forma mes/día/año y usará la clase Fecha que ahora nos es familiar. El número entero largo se usará para representar fechas en la forma `anio * 10000 + mes * 100 + dia`. Por ejemplo, usando esta representación la fecha 12/31/2000 se convierte en el número entero largo 20001231. Las fechas representadas de esta manera son muy útiles por dos razones. Primera, es posible almacenar una fecha como un solo entero, y segunda, estas fechas están en orden numéricamente creciente, facilitando en extremo su clasificación. Por ejemplo, la fecha 01/03/2002, la cual ocurre después de 12/31/2001 se convierte en el número entero 20021013, el cual es mayor que 20011231. Dado que los números enteros que representan fechas pueden exceder el tamaño de un número entero normal, los enteros siempre se declaran como largos.

Una función constructora adecuada para hacer la conversión de una fecha en número entero largo a una fecha almacenada como un mes, día y año es

```
// constructor de conversion de tipo de largo a Fecha

Fecha::Fecha(long fechaFinal)
{
  anio = int(fechaFinal/10000.0);
  mes = int((fechaFinal - anio * 10000.0)/100.0);
  dia = int(fechaFinal - anio * 10000.0 - mes * 100.0);
}
```

El programa 10.9 usa este constructor de conversión de tipo como una función de inicialización en el momento de la declaración y como un molde explícito más adelante en el programa.



Programa 10.9

```
#include <iostream>
#include <iomanip>
using namespace std;

// declaración de clase

class Fecha
{
  private:
    int mes, dia, anio;
  public:
    Fecha(int = 7, int = 4, int = 2005); // constructor
    Fecha(long);           // constructor de conversión de tipo
    void mostrarFecha();
};

// sección de implementación

// constructor
Fecha::Fecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;
}

// constructor de conversión de tipo de largo a fecha
Fecha::Fecha(long fechaFinal)
{
  anio = int(fechaFinal/10000.0);
  mes = int((fechaFinal - anio * 10000.0)/100.0);
  dia = int(fechaFinal - anio * 10000.0 - mes * 100.0);
}

// función miembro para desplegar una fecha
void Fecha::mostrarFecha()
{
  cout << setfill('0')
       << setw(2) << mes << '/'
       << setw(2) << dia << '/'
       << setw(2) << anio % 100;

  return;
}
```

(Continúa)

(Continuación)

```
int main()
{
  Fecha a, b(20061225L), c(4,1,2007); // declara 3 objetos, inicializa 2 de ellos

  cout << "Las fechas a, b y c son ";
  a.mostrarFecha();
  cout << ", ";
  b.mostrarFecha();
  cout << ", y ";
  c.mostrarFecha();
  cout << ".\n";

  a = Fecha(20080103L); // moldea un largo a fecha

  cout << "La fecha a es ahora ";
  a.mostrarFecha();
  cout << ".\n";

  return 0;
}
```

La salida producida por el programa 10.9 es

```
Las fechas a, b y c son 07/04/05, 12/25/06 y 04/01/07.
La fecha a es ahora 01/03/08.
```

El cambio en el valor de fecha de `a` ilustrado por esta salida es producido por la expresión de asignación `a = Fecha(20080103L)`, la cual usa un constructor de conversión de tipo para ejecutar el molde de largo a `Fecha`.

Conversión de clase a integrado

La conversión de un tipo de datos definido por un usuario a un tipo de datos integrado se logra usando una función operador de conversión. Una **función operador de conversión** es una función operador miembro que tiene el nombre de un tipo de datos integrado o de una clase. Cuando la función operador tiene un nombre de tipo de datos integrado se usa para convertir un tipo de datos de clase a uno integrado. Por ejemplo, una función operador de conversión para moldear un objeto de clase en un número entero largo tendría el nombre `operator long()`. Aquí el nombre de la función operador indica que tendrá lugar una conversión a un largo. Si esta función fuera parte de una clase `Fecha` se usaría para moldear un objeto `Fecha` en un número entero largo. Este uso es ilustrado por el programa 10.10.

La salida producida por el programa 10.10 es

```
La fecha de a es 04/01/07
Esta fecha, como un número entero largo, es 20070401
```

Programa 10.10

```
#include <iostream>
#include <iomanip>
using namespace std;

// declaración de clase

class Fecha
{
  private:
    int mes, dia, anio;
  public:
    Fecha(int = 7, int = 4, int = 2005);   // constructor
    operator long();         // función operador de conversión
    void mostrarFecha();
};

// sección de implementación

// constructor
Fecha::Fecha(int mm, int dd, int aaaa)
{
  mes = mm;
  dia = dd;
  anio = aaaa;
}

// función operador de conversión que convierte de Fecha a largo
Fecha::operator long() // debe devolver un entero largo
{
  long aaaammdd;

  aaaammdd = anio * 10000 + mes * 100 + dia;

  return(aaaammdd);
}

// función miembro para desplegar una fecha
void Fecha::mostrarFecha()
{
   cout << setfill('0')
        << setw(2) << mes << '/'
        << setw(2) << dia << '/'
        << setw(2) << anio % 100;
   return;
}
```

(Continúa)

(Continuación)

```
int main()
{
  Fecha a(4,1,2007); // declara e inicializa un objeto de tipo fecha
  long b;          // declara un objeto de tipo long

  b = a;          // aquí tiene lugar una conversión

  cout << "La fecha de a es ";
  a.mostrarFecha();
  cout << "\nEsta fecha, como un número entero largo, es " << b << endl;

  return 0;
}
```

El cambio en el valor de fecha de `a` a un número entero largo ilustrado por esta salida es producido por la expresión de asignación `b = a`. Esta asignación, la cual también podría haberse escrito como `b = long(a)`, llama a la función operador de conversión `long()` para que ejecute el molde de `Fecha a long`. En general, dado que la conversión explícita documenta con más claridad lo que sucede, se prefiere su uso a la de la conversión implícita.

Hay que observar que el operador de conversión no tiene argumento explícito y no tiene un tipo explícito a devolver. Esto sucede con todos los operadores de conversión: el argumento implícito siempre será un objeto de la clase que se está moldeando, y el tipo devuelto está implícito en el nombre de la función. Además, como se indicó antes, una función operador de conversión debe ser una función miembro.

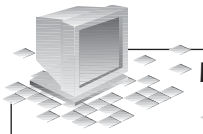
Conversión de clase a clase

Convertir de un tipo de datos de clase a un tipo de datos de clase se ejecuta de la misma manera que se moldea de un tipo de clase a un tipo de datos integrado; se hace usando una **función operador de conversión** miembro. En este caso, sin embargo, la función operador usa el nombre de la clase que se está convirtiendo en lugar de un nombre de datos integrado. Por ejemplo, si existen dos clases nombradas `Fecha` y `FechaEnt`, la función operador llamada `operator FechaEnt()` podría colocarse en la clase `Fecha` para convertir de un objeto `Fecha` a un objeto `FechaEnt`. Del mismo modo, la función operador llamada `Fecha()` podría colocarse en la clase `FechaEnt` para convertir de `FechaEnt a Fecha`.

Observe que, como antes, al convertir de un tipo de datos de clase a un tipo de datos integrado, *el nombre de la función operador determina el resultado de la conversión*; la clase que contiene la función operador determina el tipo de datos a partir del cual se hace la conversión.

Antes de proporcionar un ejemplo específico de una conversión de clase a clase, hay que señalar un punto adicional. Convertir entre clases implica que se tienen dos clases, una de las cuales siempre se define primero y una de las cuales se define en segundo lugar. Tener una función operador de conversión, dentro de la segunda clase, con el nombre de la primera clase no plantea ningún problema porque el compilador sabe de la existencia de la primera clase. Sin embargo, incluir una función operador de conversión con el nombre

de la segunda clase en la primera clase plantea un problema porque la segunda clase no ha sido definida todavía. Esto se remedia incluyendo una declaración para la segunda clase antes de la definición de la primera clase. Esta declaración, la cual se conoce de manera formal como una **declaración hacia adelante**, se ilustra en el programa 10.11, el cual también incluye operadores de conversión entre las dos clases definidas.



Programa 10.11

```
#include <iostream>
#include <iomanip>
using namespace std;

// declaración hacia delante de la clase FechaEnt
class FechaEnt;

// declaración de clase para Fecha
class Fecha
{
  private:
    int mes, dia, anio;
  public:
    Fecha(int = 7, int = 4, int = 2005); // constructor
    operator FechaEnt();     // operador de conversión Fecha a FechaEnt
    void mostrarFecha();
};
// declaración de clase para FechaEnt
class FechaEnt
{
  private:
    long aaaammdd;
  public:
    FechaEnt(long = 0);   // constructor
    operator Fecha();    // operador de conversión de fechaent a fecha
    void mostrarEnt();
};

// seccion de implementación para Fecha
Fecha::Fecha(int mm, int dd, int aaaa)  // constructor
{
  mes = mm;
  dia = dd;
  anio = aaaa;
}
// función operador de conversión que convierte de la clase Fecha a la
   clase FechaEnt
Fecha::operator FechaEnt() // debe devolver un objeto FechaEnt
```

(Continúa)

(Continuación)

```
{
  long temp;

  temp = anio * 10000 + mes * 100 + dia;
  return(FechaEnt(temp));
}
// función miembro para desplegar una Fecha
void Fecha::mostrarFecha()
{
  cout << setfill('0')
       << setw(2) << mes << '/'
       << setw(2) << dia << '/'
       << setw(2) << anio % 100;
  return;
}

// sección de implementación para FechaEnt
FechaEnt::FechaEnt(long amd) // constructor
{
  aaaammdd = amd;
}
// función operador de conversión que convierte de la clase FechaEnt a
   la clase Fecha
FechaEnt::operator Fecha()    // debe devolver un objeto Fecha
{
  int me, di, ao;

  ao = int(aaaammdd/10000.0);
  me = int((aaaammdd - ao * 10000.0)/10.0);
  di = int (aaaammdd - ao * 10000.0 - me * 100.0);
  return(Fecha(me,di,ao));
}
// función miembro para desplegar una FechaEnt
void FechaEnt::mostrarEnt()
{
  cout << aaaammdd;
  return;
}
```

```
int main()
{
  Fecha a(4,1,2007), b;     // declara dos objetos Fecha
  FechaEnt c(20081215L), d; // declara dos objetos FechaEnt

  b = Fecha(c);      // moldea c en un objeto Fecha
  d = FechaEnt(a);  // moldea a en un objeto FechaEnt
```

(Continúa)

(Continuación)

```
   cout << "La fecha a es ";
   a.mostrarFecha();
   cout << "\n como un objeto FechaEnt esta fecha es ";
   d.mostrarEnt();

   cout << "\n La fecha c es ";
   c.mostrarEnt();
   cout << "\n como un objeto Fecha esta fecha es ";
   b.mostrarFecha();
   cout << endl;

   return 0;
}
```

La salida producida por el programa 10.11 es

```
La fecha a es 04/01/07
  como un objeto FechaEnt esta fecha es 20070401
La fecha c es 20081215
  como un objeto Fecha esta fecha es 12/15/08
```

Como lo ilustra el programa 10.11, el molde de `Fecha a FechaEnt` es producido por la asignación `b = Fecha(c)` y el molde de `FechaEnt a Fecha` es producido por la asignación `d = FechaEnt(a)`. De manera alternativa, las asignaciones `b = c` y `d = a` producirían los mismos resultados. Observe también la declaración hacia delante de la clase `FechaEnt` antes de la declaración de la clase `Fecha`. Esto es necesario para que la clase `Fecha` pueda hacer referencia a `FechaEnt` en su función operador de conversión.

Ejercicios 10.4

1. **a.** Defina las cuatro conversiones de tipos de datos disponibles en C++ y el método para lograr cada conversión.
   b. Defina los términos constructor de conversión de tipo y función operador de conversión y describa cómo se usan en conversiones definidas por el usuario.

2. Escriba un programa en C++ que declare una clase llamada `Tiempo` que tenga miembros de datos en número entero llamados `horas, minutos` y `segundos`. Incluya en el programa un constructor de conversión de tipo que convierta un número entero largo, que represente los segundos transcurridos desde medianoche, en una representación equivalente como horas:minutos:segundos. Por ejemplo, el número entero largo 30336 sería convertido al tiempo 8:25:36. Use una representación militar del tiempo de modo que 2:30 pm sea representado como 14:30:00. La relación entre representaciones de tiempo es

   segundos transcurridos = horas * 3600 + minutos * 60 + segundos.

3. Una fecha juliana se representa como el número de días desde una fecha base conocida. Un algoritmo para convertir desde una fecha gregoriana, en la forma mes/día/año, a una fecha juliana con una fecha base de 00/00/0000 se proporciona a continuación. Todos los cálculos en este algoritmo usan aritmética de números enteros, lo cual significa que debe desecharse la parte fraccionaria de todas las divisiones. En este algoritmo M = mes, D = día y A = año.

   Si M es menor que o igual a 2
   Establecer las variables MP = 0 y AP = A-1
   De no ser así
   Establecer MP = int(0.4 * M + 2.3) y AP = A
   Termina el si
   T = int(AP/4) – int(AP/100) + int(AP/400)
   Fecha juliana = 365 * A + 31 * (M – 1) + D + T – MP

   Usando este algoritmo, modifique el programa 10.10 para moldear desde un objeto de fecha gregoriana a su correspondiente representación juliana como un número entero largo. Pruebe su programa usando las fechas gregorianas 1/31/2005 y 3/16/2006, las cuales corresponden a las fechas julianas 732342 y 732751, respectivamente.

4. Modifique el programa escrito para el ejercicio 2 para incluir una función operador de conversión miembro que convierte un objeto de tipo `Tiempo` en un número entero largo que represente el número de segundos a partir de medianoche.

5. Escriba un programa en C++ que tenga una clase `Fecha` y una clase `Juliana`. La clase `Fecha` deberá ser la misma clase `Fecha` que se usó en el programa 10.11, mientras la clase `Juliana` deberá representar una fecha como un número entero largo. Para este programa incluya una función operador de conversión miembro dentro de la clase `Fecha` que convierta un objeto `Fecha` en un objeto `Juliana`, usando el algoritmo presentado en el ejercicio 3. Pruebe su programa convirtiendo las fechas 1/31/2006 y 3/16/2007, las cuales corresponden a las fechas julianas 732707 y 733116, respectivamente.

6. Escriba un programa en C++ que tenga una clase `Tiempo` y una clase `TiempoL`. La clase `Tiempo` deberá tener miembros de datos enteros llamados `horas`, `minutos` y `segundos`, mientras la clase `TiempoL` deberá tener un miembro de datos largo llamado `seg_trans`, el cual representa el número de segundos transcurridos desde medianoche. Para la clase `Tiempo` incluya una función operador de conversión miembro llamada `TiempoL()` que convierta un objeto `Tiempo` en un objeto `TiempoL`. Para la clase `TiempoL` incluya una función operador de conversión miembro llamada `Tiempo()` que convierta un objeto `TiempoL` en un objeto `Tiempo`.

10.5 APLICACIÓN: SIMULACIÓN DE UNA BOMBA DE GASOLINA CON OBJETOS MÚLTIPLES

En la aplicación 2 de la sección 9.3 se hizo un requerimiento para construir un programa en C++ que simulaba la operación de una bomba de gasolina durante el transcurso de cualquier periodo de 30 minutos. Por conveniencia, este requerimiento se repite a continuación:

Se ha solicitado que escribamos un programa que simule la operación de una bomba de gasolina. En cualquier momento durante la simulación, deberíamos poder determinar, a partir de la bomba, el precio por galón de gasolina y la cantidad restante en el tanque

de suministro desde el que se extrae la gasolina. Si la cantidad de gasolina en el tanque de suministro es mayor que o igual a la cantidad de gasolina solicitada, la solicitud debe despacharse; de lo contrario, sólo deberá usarse la cantidad disponible en el tanque de suministro. Una vez que se ha bombeado la gasolina, deberá desplegarse el precio total de los galones surtidos y la cantidad de gasolina en galones que se ha bombeado deberá restarse de la cantidad en el tanque de suministro.

Para la simulación, suponga que la bomba está inactiva en forma aleatoria entre 1 a 15 minutos entre llegadas de clientes y que un cliente solicita en forma aleatoria entre 3 y 20 galones de gasolina. Aunque la capacidad del tanque de suministro por omisión es de 500 galones, suponga que la cantidad inicial de gasolina en el tanque para esta simulación es de sólo 300 galones. Al principio, el programa deberá simular un lapso de media hora.

Además, para cada llegada de un cliente y solicitud de gasolina, se desea saber el tiempo de inactividad antes de la llegada del cliente, cuántos galones de gasolina fueron surtidos y el precio total de la transacción. La bomba en sí deberá mantenerse al tanto del precio por galón de gasolina y la cantidad de gasolina restante en el tanque de suministro. Por lo general, el precio por galón es $1.80, pero el precio para la simulación deberá ser $2.00.

Habiendo construido una clase `Bomba` para modelar la operación de una bomba de gasolina, ésta puede utilizarse dentro del contexto de una simulación completa. Se hará proporcionando una clase `Cliente` y luego controlando la interacción entre estas dos clases usando una función main() para crear un programa real de simulación de clases múltiples.

Paso 1 *Analizar el problema*

Como se especificó, el problema implica dos tipos de objetos: una bomba de gas y un cliente. Considérense estos tipos de objeto por separado.

La bomba. Una clase `Bomba` se diseñó e implementó en la sección 9.3 y se repite aquí por comodidad:

```
#include <iostream>
#include <iomanip>
using namespace std;

const double CANTIDAD_EN_TANQUE = 500; // galones iniciales en el tanque
const double PRECIO_OMISION = 1.80;   // precio por galón

class Bomba
{
  // sección de declaración de datos
  private:
    double cantEnTanque;
    double precio;

  // declaraciones de métodos
  public:
    Bomba(double = PRECIO_OMISION, double = CANTIDAD_EN_TANQUE); // constructor
    void obtenerValores();
    void solicitud(double);
};
```

```
// sección de implementación de métodos


Bomba::Bomba(double precioHoy, double cantidadEnTanque)
{
  cantEnTanque = cantidadEnTanque;
  precio = precioHoy;
}

void Bomba::obtenerValores()
{
  cout << "El tanque de gasolina tiene " << cantEnTanque << " galones de
          gasolina." << endl;
  cout << "El precio por galón de gasolina es $" << setiosflags
          (ios::showpoint)
       << setprecision(2) << setiosflags(ios::fixed) << precio << endl;
}

void Bomba::solicitud(double cantSurt)
{
  double surtida;

  if (cantEnTanque >= cantSurt)
     surtida = cantSurt;
  else
   surtida = cantEnTanque;

  cantEnTanque -= surtida;
  cout << cantSurt << " galones fueron solicitados " << endl;
  cout << surtida << " galones fueron surtidos" << endl;
  cout << cantEnTanque << " galones quedan en el tanque" << endl;
  cout << "El precio total es $" << setiosflags(ios::showpoint)
       << setprecision(2) << (surtida * precio) << endl;

  return;
}
```

Deberá revisar la aplicación 2 en la sección 9.3 si los métodos de datos o de miembros de esta clase `Bomba` no son claros. Para su comodidad posterior al escribir el programa de simulación requerido, suponga que el código para la clase `Bomba` se ha nombrado `Bomba.cpp` y se ha almacenado dentro de una carpeta, `cpcode`, en la unidad `C`. Una vez hecho esto, puede incluirse la definición de la clase `Bomba` en un programa usando la directiva preprocesadora de una sola línea:

```
#include <C:\\cpcode\\Bomba.cpp>.
```

[8]

[8] La razón para usar un nombre de ruta completo en la instrucción `#include` es asegurar que el preprocesador tenga acceso a la clase `Bomba.cpp` que se ha colocado en la carpeta nombrada. No se desea que el preprocesador busque su carpeta por omisión y tal vez localice alguna otra clase `Bomba.cpp` de la que no sabemos o sea incapaz de localizar los archivos `Bomba.cpp` en absoluto.

El cliente. Para esta simulación, hay múltiples instancias de clientes que llegan en forma aleatoria entre 1 y 15 minutos y solicitan gasolina en cantidades que varían entre 3 y 20 galones. Desde un punto de vista del objeto, sin embargo, no nos interesa almacenar la llegada y el número de galones solicitados por cada cliente. Se necesita un objeto cliente que nos presente un tiempo de llegada y una solicitud de gasolina en galones. Por consiguiente, el tipo de objeto `Cliente` no necesita tener atributos pero debe proporcionar dos operaciones. La primera operación, `llegada()`, proporcionará un tiempo de llegada aleatorio entre 1 y 15 minutos. La segunda operación, `galones()`, proporcionará una solicitud aleatoria de entre 3 y 20 galones de gasolina. Esta clase es simple y su diseño e implementación se codifican como sigue:

```
#include <ctime>
#include <cmath>
using namespace std;

//declaración e implementación de clase
class Cliente
{
  public:
    Cliente() {srand(tiempo(NULL));};
    int llegada() {return(1 + rand() % 15);};
    int galones() {return(3 + rand() % 18);};
};
```

El constructor de clase se usa para aleatorizar la función `rand()`. Repase la sección 6.7 si no está familiarizado con las funciones `srand()` o `rand().)` La función `llegada()` devuelve un número entero aleatorio entre 1 y 15, y la función `galones()` devuelve un número entero aleatorio entre 3 y 20. (Se le dejará como ejercicio volver a escribir la función `galones()` para devolver un valor que no sea entero.) Dado que todas las funciones son de una sola línea, se han incluido sus definiciones dentro de la sección de declaración como funciones en línea.

Una vez más, por comodidad posterior al escribir el programa de simulación completo, suponga que el código para la clase `Cliente` se coloca en un archivo llamado `Cliente.cpp` dentro de una carpeta llamada `cpcode` en la unidad C. Una vez hecho esto, incluir la clase `Cliente` dentro de un programa requiere la siguiente instrucción:

```
#include <C:\\cpcode\\Cliente.cpp>
```

Lógica del programa. Habiendo analizado y definido las dos clases que se usarán, es necesario analizar y definir la lógica para controlar la interacción entre los objetos `Cliente` y `Bomba` para una simulación válida. En este caso particular, la única interacción entre un objeto `Cliente` y un objeto `Bomba` es que una llegada de un cliente, seguida por una solicitud, determina cuándo se activa la `Bomba` y cuánta gasolina se solicita. Por tanto, cada interacción entre un `Cliente` y una `Bomba` puede expresarse con el siguiente seudocódigo:

Obtener un tiempo de llegada del Cliente
Obtener una solicitud de gasolina del Cliente
Activar la Bomba con la solicitud

Aunque esta repetición de eventos tiene lugar en forma continua en el transcurso de un día, nos interesa un periodo de media hora. Por consiguiente, estos tres eventos deben colocarse en un ciclo que se ejecuta hasta que ha transcurrido el tiempo requerido por la simulación.

Punto de información

Bibliotecas de programa y de clase

El concepto de biblioteca de programa comenzó con FORTRAN, el cual fue el primer lenguaje comercial de alto nivel introducido en 120054. La biblioteca FORTRAN consistía en un grupo de rutinas matemáticas probadas y depuradas por completo proporcionadas con el compilador. Desde entonces, todo lenguaje de programación ha proporcionado su propia biblioteca de funciones. En C y C++, esta biblioteca se conoce como la biblioteca de programa estándar e incluye más de 12 000 funciones declaradas en quince archivos de encabezado diferentes. Los ejemplos de funciones de biblioteca estándar incluyen `sqrt()`, `pow()`, `abs()`, `rand()`, `srand()` y `time()`. La ventaja de las funciones de biblioteca es que mejoran el desarrollo y diseño del programa al proporcionar código que se sabe que funciona sin la necesidad de pruebas y depuración adicionales.

Con la introducción de lenguajes orientados a objetos, el concepto de biblioteca de programa se ha extendido para incluir bibliotecas de clase. Una *biblioteca de clase* es una biblioteca de clases probadas y depuradas.

Una de las características prácticas clave de las bibliotecas de clase es que ayudan reutilizar código en forma significativa. Al proporcionar código probado y depurado consistente en miembros de datos y funciones, las bibliotecas de clase suministran grandes secciones de código preescrito y reutilizable listo para su incorporación dentro de aplicaciones nuevas. Esto cambia el enfoque de escribir programas de aplicación a partir de la creación de código nuevo, a entender cómo usar objetos predefinidos y unirlos en una forma coherente y útil.

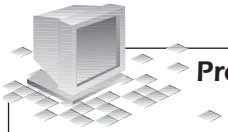
Paso 2 Desarrollar una solución

Una vez desarrolladas y codificadas las dos clases requeridas, `Bomba` y `Cliente`, falta desarrollar la lógica de control dentro de la función `main()` para activar eventos de clase. Esto requerirá un ciclo controlado por el tiempo de llegada total para todos los clientes. Una estructura de control adecuada para `main()` se describe con el siguiente algoritmo:

Crear un objeto Bomba con el suministro de gasolina inicial requerido
Desplegar los valores en la `Bomba` *inicializada*
Establecer el tiempo total en 0
Obtener un tiempo de llegada del Cliente // primera llegada
Sumar el tiempo de llegada al tiempo total
Mientras el tiempo total no exceda el tiempo de la simulación
    Desplegar el tiempo total
    Obtener una solicitud de gasolina del Cliente
    Activar la `Bomba` *con la solicitud*
    Obtener un tiempo de llegada del Cliente // siguiente llegada
    Sumar el tiempo de llegada al tiempo total
Termina while
Desplegar un mensaje indicando que la simulación terminó

Paso 3 Codificar la solución

El código C++ correspondiente al diseño se muestra como en el programa 10.12.



Programa 10.12

```
#include <C:\\cpcode\\Bomba.cpp> // nótese aquí el uso del nombre de la ruta completo
#include <C:\\cpcode\\Cliente.cpp // de nuevo se usa un nombre de ruta completo

const double TIEMPOSIM = .5;          // tiempo de la simulación en horas
const int MINUTOS = 60;               // número de minutos en una hora

int main()
{
  Bomba a(2.00, 300); // declara 1 objeto de tipo Bomba
  Cliente b;          // declare 1 object of type Customer
  int tiempoTotal = 0;
  int tiempoInact;
  int cantSolicitada;
  int MinutosSim;  // tiempo de la simulación en minutos


  MinutosSim = TIEMPOSIM * MINUTOS;
  cout << "\nInicio de una simulación nueva, el tiempo de simulación es "
       << MinutosSim << " minutos " << endl << endl;
  a.obtenerValores();

  // obtener la primera llegada
  tiempoInact = b.llegada();
  tiempoTotal += tiempoInact;

  while (tiempoTotal <= MinutosSim)
  {
    cout << "\nEl tiempo inactiva es " << tiempoInact << " minutos" << endl
       << " y llevamos " << tiempoTotal
       << " minutos en la simulación." << endl;
    cantSolicitada = b.galones();
    a.solicitada(double(cantSolicitada));

    // obtener la siguiente llegada
    tiempoInact = b.llegada();
    tiempoTotal += tiempoInact;
  }
  cout << "\nEl tiempo inactiva es " << tiempoInact << " minutos." << endl;
      << "Como el tiempo total ahora excede el tiempo de la simulación, " << endl
       << " este ejercicio de simulación termina." << endl;

  return 0;
}
```

Paso 4 Prueba y corrección del programa

Suponiendo que las clases `Bomba` y `Cliente` se han probado y depurado en forma minuciosa, probar y depurar el programa 10.12 se restringe a probar y depurar el método `main()`. Esta especificidad de la prueba es precisamente una de las ventajas del enfoque orientado a objetos. Usar definiciones de clase escritas y probadas con anterioridad permite enfocar la atención en el código restante que controla el flujo de eventos entre objetos, lo cual en el programa 10.12 se centra en el método `main()`.

En sí mismo, el método `main()` en el programa 10.12 es un ciclo `while` sencillo donde el tiempo en que está inactiva la `Bomba` corresponde al tiempo entre llegadas de clientes. La salida de una muestra de la ejecución, que se muestra a continuación, verifica que el ciclo esté operando en forma correcta:

```
Inicio de una simulación nueva, el tiempo de simulación es
30 minutos

El tanque de gasolina tiene 300 galones de gasolina.
El precio por galón de gasolina es $2.00

El tiempo inactiva es 2 minutos
   y llevamos 2 minutos en la simulación.
7.00 galones fueron solicitados
7.00 galones fueron surtidos
293.00 galones quedan en el tanque
El precio total es $14.00

El tiempo inactiva es 1 minutos
   y llevamos 3 minutos en la simulación.
15.00 galones fueron solicitados
15.00 galones fueron surtidos
278.00 galones quedan en el tanque
El precio total es $30.00

El tiempo inactiva es 11 minutos
   y llevamos 14 minutos en la simulación.
13.00 galones fueron solicitados
13.00 galones fueron surtidos
265.00 galones quedan en el tanque
El precio total es $26.00

El tiempo inactiva es 8 minutos
   y llevamos 22 minutos en la simulación.
20.00 galones fueron solicitados
20.00 galones fueron surtidos
245.00 galones quedan en el tanque
El precio total es $40.00

El tiempo inactiva es 9 minutos.
Como el tiempo total ahora excede el tiempo de la simulación,
   este ejercicio de simulación termina.
```

Ejercicios 10.5

1. Introduzca el programa 10.12 en su computadora y ejecútelo.
2. **a.** Elimine las funciones `inline` en las secciones de declaración e implementación de la clase `Cliente`. Discuta cuál forma de la clase `Cliente` prefiere y por qué.
   b. Vuelva a escribir la función `galones()` en la clase `Cliente` de modo que devuelva un número en punto flotante entre 3.0 y 20.0 galones.
3. En lugar de la función `main()` usada en el programa 10.12, un estudiante propuso lo siguiente:

```
int main()
{
  Bomba a(CANT_EN_TANQUE, PRECIO_HOY); // declara 1 objeto de tipo Bomba
  Cliente b;                           // declara 1 objeto de tipo Cliente
  int tiempoTotal = 0;
  int tiempoInact;
  int cantSolicitada;
  int MinutosSim;   // tiempo de la simulación en minutos

  MinutosSim = TIEMPOSIM * MINUTOS;
  cout << "\nInicio de una simulación nueva, el tiempo de simulación es "
       << MinutosSim << " minutos" << endl;
  a.valores();

  do
  {
    tiempoInact = b.llegada();
    tiempoTotal += tiempoInact;
    if (tiempoTotal > (TIEMPOSIM * MINUTOS))
    {
      cout << "\n El tiempo inactiva es " << tiempoInact << " minutos." << endl
           << "Como el tiempo total ahora excede el tiempo de la
               simulación, " << endl
           << "este ejercicio de simulación termina." << endl;
      break;
    }
    else
    {
      cout << "El tiempo inactiva es " << tiempoInact << " minutos" << endl;
         << "   y llevamos " << tiempoTotal
         << " minutos en la simulación." << endl;
      cantSolicitada = b.galones();
      a.solicitada(double(cantSolicitada));
    }
  } while (1); // siempre verdadero

  return 0;
}
```

Determine si esta función `main()` produce una simulación válida. Si no lo hace, discuta por qué no. Si lo hace, discuta cuál versión prefiere y por qué.

4. Construya una clase llamada `Elevador` que pueda usarse para crear un objeto `elevador`. Para este problema, el único atributo de interés es la ubicación del elevador. Represente esta ubicación, la cual corresponde al piso actual en que se encuentra el elevador, con una variable en número entero llamada `pisoActual`. Las funciones que deberán proporcionarse para esta clase son un constructor para establecer el piso de la posición inicial de un objeto `elevador` nuevo en el primer piso y una función de mutación para cambiar la posición (estado) del elevador a un piso nuevo. Aquí está el algoritmo que describe la función de mutación:

```
Si se hace una solicitud para un piso arriba de 15 o para el piso actual,
     No hacer nada
De lo contrario si la solicitud es para un piso arriba del piso actual
    Desplegar el número de piso actual
    Mientras no esté en el piso designado
        Incrementar el número de piso
        Desplegar el nuevo número de piso
    Termina mientras
    Desplegar el número de piso final
De lo contrario      // la solicitud debe ser para un piso debajo del piso actual
    Desplegar el número de piso actual
    Mientras no esté en el piso designado
        Disminuir el número de piso
        Desplegar el nuevo número de piso
    Termina mientras
    Desplegar el número de piso final
Termina el si
```

Este algoritmo consiste de una instrucción `if-else` que tiene tres partes: si se solicita un servicio incorrecto, no se toma ninguna acción; si se selecciona un piso arriba de la posición actual, el elevador sube; y si se selecciona un piso abajo de la posición actual, el elevador baja. Para el movimiento hacia arriba y abajo, puede usarse un ciclo `while` para incrementar o disminuir la posición del elevador un piso a la vez y reportar el movimiento usando una instrucción `cout`.

5. Usando la clase `Elevador` definida en el ejercicio 4 y definiendo una clase nueva llamada `Persona`, construya una simulación por la cual una persona llega al azar en cualquier momento de 1 a 10 minutos en cualquier piso y llame al elevador. Si el elevador no está en el piso en que está la persona, debe moverse al piso en que está la persona. Una vez dentro del elevador la persona puede seleccionar cualquier piso. Ejecute la simulación para tres personas que lleguen al azar y haga que la simulación despliegue el movimiento del elevador.

10.6 HERENCIA DE CLASE

La capacidad para crear clases nuevas a partir de otras existentes es la motivación y poder subyacente detrás de las técnicas de programación orientadas a clases y a objetos. Hacer esto facilita reutilizar código existente en formas nuevas sin necesidad de volver a hacer los procesos de prueba y validación. Permite a los diseñadores de una clase ponerla a disposición de otros para adiciones y extensiones sin renunciar al control sobre las características existentes de la clase.

Construir una clase a partir de otra se logra usando una capacidad llamada herencia. Relacionada con esta capacidad hay una característica igual de importante llamada polimorfismo. El polimorfismo proporciona la capacidad de redefinir la forma en que operan las funciones miembro de clases relacionadas basada en el objeto de la clase al que se está haciendo referencia. De hecho, para que un lenguaje de programación sea clasificado como un lenguaje orientado a objetos debe proporcionar las características de clases, herencia y polimorfismo. En esta sección se describen las características de herencia y polimorfismo proporcionadas en C++.

La **herencia** es la capacidad de derivar una clase a partir de otra. La clase inicial usada como base para la clase derivada se conoce como **base, padre** o **superclase**. La clase derivada se conoce como **derivada, hija** o **subclase**.

Una clase derivada es una clase nueva por completo que incorpora todos los datos y funciones miembro de su clase base. Sin embargo, puede agregar sus propios datos y miembros de funciones nuevos adicionales y puede anular cualquier función de clase base, y por lo general lo hace.

Como ejemplo de herencia, considérense tres formas geométricas, un círculo, un cilindro y una esfera. Todas estas formas comparten una característica común, un radio. Por tanto, para estas formas puede hacerse al círculo un tipo base para las otras dos formas, como se ilustra en la figura 10.5. Por convención, las flechas siempre apuntan de la clase derivada a la clase base. Al reformular estas formas como tipos de clase, se haría al círculo la clase base y las clases cilindro y esfera se derivarían de ella.

Las relaciones ilustradas en la figura 10.5 son ejemplos de herencia simple. En la **herencia simple** cada tipo derivado sólo tiene un tipo base inmediato. El complemento de la herencia simple es la herencia múltiple. En la **herencia múltiple** un tipo derivado tiene dos o más tipos base. La figura 10.6 ilustra un ejemplo de herencia múltiple. En este texto sólo se considerará la herencia simple.

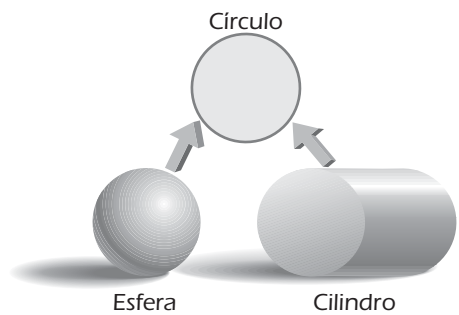


Figura 10.5 *Relación de tipos de objeto.*

Punto de información

Lenguajes basados en objetos frente a orientados a objetos

Un lenguaje ***basado* en objetos** es aquel en el cual los datos y las operaciones pueden incorporarse de tal forma que los valores de datos pueden aislarse y tenerse acceso a ellos a través de las funciones de clase especificadas. La capacidad para vincular los miembros de datos con operaciones en una sola unidad se conoce como **encapsulamiento**. En C++ el encapsulamiento es proporcionado por su capacidad de clase.

Para que un lenguaje sea clasificado como ***orientado a objetos*** debe proporcionar también herencia y polimorfismo. La ***herencia*** es la capacidad para derivar una clase de otra. Una clase derivada es un tipo de datos nuevo por completo que incorpora todos los miembros de datos y funciones miembro de la clase original con los datos y miembros de funciones nuevos únicos en sí mismos. La clase usada como la base para el tipo derivado se conoce como clase ***base*** o ***padre*** y el tipo de datos derivado se conoce como clase ***derivada*** o ***hija***.

El **polimorfismo** permite que el mismo nombre de método invoque una operación en objetos de una clase padre y una operación diferente en objetos de una clase derivada.

C++, el cual proporciona encapsulamiento, herencia y polimorfismo, es un lenguaje orientado a objetos verdadero. Debido a que C, el cual es predecesor de C++, no proporciona estas características, no es un lenguaje basado en objetos ni uno orientado a objetos.

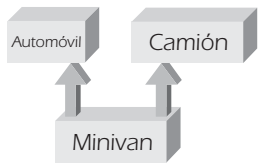


Figura 10.6 *Un ejemplo de herencia múltiple.*

Las derivaciones de clase ilustradas en las figuras 10.5 y 10.6 se conocen de manera formal como ***jerarquías de clase*** porque ilustran la jerarquía, u orden, en el que una clase se deriva de otra. Ahora se verá cómo derivar una clase de otra.

Una clase derivada tiene la misma forma que cualquier otra clase en cuanto a que consiste de una declaración y una implementación. La única diferencia está en la primera línea de la sección de declaración. Para una clase derivada, esta línea se extiende para incluir una especificación de acceso y un nombre de clase base y tiene la forma:

clase derivadaNombreClase : accesoClase nombreClaseBase

Por ejemplo, si `Circulo` es el nombre de una clase existente, una clase nueva llamada `Cilindro` puede derivarse como sigue:

```
class Cilindro : public Circulo
{
    // aquí se agregan los datos adicionales y
    // miembros de función
};  // fin de la declaración de la clase Cilindro
```

Excepto para el especificador de acceso a la clase después de los dos puntos y el nombre de la clase base, no hay nada inherentemente nuevo o complicado acerca de la construcción de la clase `Cilindro`. Antes de proporcionar una descripción de la clase `Circulo` y agregar datos y miembros de función a la clase derivada `Cilindro`, es necesario reexaminar los especificadores de acceso y cómo se relacionan con clases derivadas.

Especificaciones de acceso

Hasta ahora sólo se han usado especificadores de acceso privados y públicos dentro de una clase. Dar categoría de privados a todos los miembros de datos aseguraba que sólo podían tener acceso a ellos las funciones miembro o amigas de clase. Este acceso restringido impide el acceso a las funciones que no sean de la clase (excepto amigas), lo cual *también impide el acceso a las funciones de clase derivada*. Ésta es una restricción sensible porque si no existiera cualquiera podría "saltarse" la restricción privada con sólo derivar una clase.

Para conservar un tipo de acceso restringido a través de clases derivadas, C++ proporciona una tercera especificación de acceso: protegida. Un acceso protegido se comporta en forma idéntica al acceso privado en cuanto a que sólo permite el acceso a las funciones miembro o amigas, pero permite que esta restricción sea heredada por cualquier clase derivada. Entonces la clase derivada define el tipo de herencia que está dispuesto a tomar, sujeta a las restricciones de acceso de la clase base. Esto se hace con el especificador de acceso a la clase, el cual aparece después de los dos puntos al principio de su sección de declaración. La tabla 10.3 muestra el acceso al miembro de la clase derivada resultante basado en las especificaciones de los miembros de la clase base y el especificador de acceso de la clase derivada.

Tabla 10.3 Restricciones de acceso heredadas

Miembro de la clase base	Acceso de la clase derivada	Miembro de la clase derivada
`private`	`: private`	`inaccesible`
`protected`	`: private`	`privado`
`public`	`: private`	`privado`
`private`	`: public`	`inaccesible`
`protected`	`: public`	`protegido`
`public`	`: public`	`público`
`private`	`: protected`	`inaccesible`
`protected`	`: protected`	`protegido`
`public`	`: protected`	`protegido`

Usando la tabla 10.3 puede verse (región sombreada) que si el miembro de la clase base tiene un acceso protegido y el especificador de la clase derivada es `public`, entonces el miembro de la clase derivada estará protegido de su clase. Del mismo modo, si la clase base tiene un acceso público y el especificador de la clase derivada es `public`, el miembro de la clase derivada será público. Como éste es el tipo de especificación más usado para los datos de la clase base y los miembros de función, respectivamente, es el que se usará aquí. Esto significa que para todas las clases que pretendan utilizarse como una clase base se usará un acceso `protected` a los miembros de datos en lugar de una designación `private`.

Un ejemplo

Para ilustrar el proceso de derivar una clase de otra se derivará una clase `Cilindro` de una clase base `Circulo`. La definición de la clase `Circulo` es:

```
// declaración de la clase

class Circulo
{
  protected:
    double radio;
  public:
    Circulo(double = 1.0);  // constructor
    double calcval();
};

// implementación de la clase

// constructor
Circulo::Circulo(double r) // constructor
{
  radio = r;
}

// calcula el área de un círculo
double Circulo::calcval()
{
  return(PI * radio * radio);
}
```

Excepto por la sustitución del especificador de acceso `protected` en lugar del especificador `private` usual para los miembros de datos, la cual es una definición de clase estándar. La única variable no definida es `PI`, la cual se usa en la función `calcval()`. Ésta se definirá como

```
const double PI = 2.0 * asin(1.0);
```

Éste es tan sólo un "truco" que obliga a la computadora a devolver el valor de `PI` con una precisión de tantos lugares decimales como lo permita la computadora. Este valor se obtiene tomando el seno del arco de 1.0, el cual es $\pi/2$, y multiplicando el resultado por 2.

Habiendo definido la clase base, ahora puede extenderse a una clase derivada. La definición de la clase derivada es

```
// declaración de clase donde
// Cilindro se deriva de Circulo

class Cilindro : public Circulo
{
  protected:
    double largo; // agrega un miembro de datos adicional y
  public: // dos miembros de función adicionales
```

```
  Cilindro(double r = 1.0, double l = 1.0) : Circulo(r), largo(l) {}
  double calcval();
};

// implementación de clase

double Cilindro::calcval() // esto calcula un volumen
{
  return (largo * Circulo::calcval()); // nótese la llamada a
                                       la función base
}
```

Esta definición abarca varios conceptos importantes relacionados con las clases derivadas. Primero, como una clase derivada, `Cilindro` contiene todos los datos y miembros de función de su clase base, `Circulo`, más los miembros adicionales que pueda agregar. En este caso particular la clase `Cilindro` consiste de un miembro de datos radio, heredada de la clase `Circulo`, además de un miembro largo adicional. Por tanto, cada objeto `Cilindro` contiene dos miembros de datos, como se ilustra en la figura 10.7.

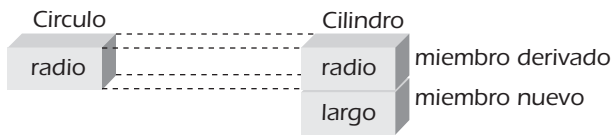


Figura 10.7 *Relación entre miembros de datos* `circulo` *y* `cilindro`.

Además de tener dos miembros de datos, la clase `Cilindro` también hereda los miembros de la función `Circulo`. Esto se ilustra en el constructor `Cilindro`, el cual usa una lista de inicialización de miembros base (véase la sección 10.1) que de manera específica llama al constructor `Circulo`. También se ilustra en la función `calcval() de Cilindro`, la cual hace una llamada a `Circulo::calcval()`.

En ambas clases se ha usado específicamente el mismo nombre de la función, `calcval()`, para ilustrar la anulación de una función base por una función derivada. Cuando un objeto `Cilindro llama a calcval()` es una solicitud para usar la versión `Cilindro` de la función, mientras la llamada de un objeto `Circulo a calcval()` es una solicitud para usar la versión `Circulo`. En este caso la clase `Cilindro` sólo puede tener acceso la versión de clase de `calcval()` usando el operador de resolución de alcance, como se hace en la llamada `Circulo::calcval()`. El programa 10.13 usa estas dos clases dentro del contexto de un programa completo. La salida producida por el programa 10.13 es

```
El área del círculo_1 es 3.14159
El área del círculo_2 es 12.5664
El volumen del cilindro_1 es 113.097

El área del círculo_1 ahora es 28.2743
```

Programa 10.13

```
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 2.0 * asin(1.0);

// declaración de la clase

class Circulo
{
  protected:
    double radio;
  public:
    Circulo(double = 1.0);  // constructor
    double calcval();
};
// sección de implementación de la clase

// constructor
Circulo::Circulo(double r)
{
  radio = r;
}

// calcula el área de un círculo
double Circulo::calcval()
{
  return(PI * radio * radio);
}

// declaración de clase para la clase derivada
// Cilindro que se deriva de Circulo
class Cilindro : public Circulo
{
  protected:
    double largo;   // agrega un miembro de datos adicional y
  public:           // dos miembros de función adicionales
    Cilindro(double r = 1.0, double l = 1.0) : Circulo(r), largo(l) {}
    double calcval();
};
```

(Continúa)

(Continuación)

```
// sección de implementación para Cilindro

double Cilindro::calcval()   // esto calcula un volumen
{
  return (largo * Circulo::calcval()); // nótese la llamada a la función base
}

int main()
{
  Circulo circulo_1, circulo_2(2); // crea dos objetos Circulo
  Cilindro cilindro_1(3,4);        // crea un objeto Cilindro

  cout << "El área del circulo_1 es " << circulo_1.calcval() << endl;
  cout << "El área del circulo_2 es " << circulo_2.calcval() << endl;
  cout << "El volumen del cilindro_1 es " << cilindro_1.calcval() << endl;

  circulo_1 = cilindro_1; // asigna un cilindro a un Circulo

  cout << "\nEl área del circulo_1 ahora es " << circulo_1.calcval() << endl;

  return 0;
}
```

Las primeras tres líneas de salida son todas sencillas y son producidas por las primeras tres instrucciones `cout` en el programa. Como muestra la salida, una llamada a `calcval()` usando un objeto `Circulo` activa la versión `Circulo` de esta función, mientras una llamada a `calcval()` usando un objeto `Cilindro` activa la versión `Cilindro`.

La instrucción de asignación `circulo_1 = cilindro_1;` introduce otra relación importante entre una clase base y una derivada: *un objeto de clase derivada puede ser asignado a un objeto de clase base*. Esto no debería sorprender porque tanto la clase base como la derivada comparten un conjunto común de tipos de miembro de datos. En este tipo de asignación sólo se asigna este conjunto de miembros de datos, el cual consiste de todos los miembros de datos de la clase base. Por tanto, como se ilustra en la figura 10.8, la asignación de `Cilindro` a `Circulo` produce la siguiente asignación miembro por miembro:

```
circulo_1.radio = cilindro_1.radio;
```

El miembro largo del objeto `Cilindro` no se usa en la asignación debido a que no tiene una variable equivalente en la clase `Circulo`. El molde inverso, de clase base a derivada, no es tan simple y requiere un constructor para inicializar en forma correcta los miembros adicionales de la clase derivada que no están en la clase base.

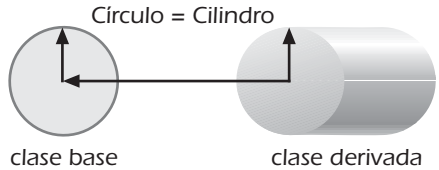


Figura 10.8 *Asignación de clase derivada a clase base.*

Antes de dejar el programa 10.13, hay que señalar un punto adicional. Aunque el constructor `Circulo` fue llamado de manera explícita usando una lista de inicialización base/miembro para el constructor `Cilindro`, también podría haberse hecho una llamada implícita. En ausencia de un constructor de clase derivada explícito el compilador llamará primero de manera automática al constructor de la clase base por omisión, antes que se llame al constructor de la clase derivada. Esto funciona porque la clase derivada contiene todos los miembros de datos de la clase base. De un modo similar las funciones destructoras son llamadas en el orden inverso, primero la clase derivada y luego la clase base.

Ejercicios 10.6

1. Defina los siguientes términos:
   - **a.** herencia
   - **b.** clase base
   - **c.** clase derivada
   - **d.** herencia simple
   - **e.** herencia múltiple
   - **f.** jerarquía de clase

2. Describa la diferencia entre un miembro de clase `private` y uno `protected`.

3. ¿Cuáles tres características debe proporcionar un lenguaje de programación para que se clasifique como un lenguaje orientado a objetos?

4. **a.** Modifique el programa 10.13 para incluir una clase derivada llamada `Esfera` a partir de la clase base `Circulo`. Los únicos miembros de clase adicionales de `Esfera` deberán ser un constructor y una función `calcval()` que devuelva el volumen de la esfera. (NOTA: *volumen = 4/3 π radio*^{3})
   b. Incluya la clase construida para el ejercicio 4a en un programa funcional en C++. Haga que su programa llame a todas las funciones miembro en la clase `Esfera`.

5. **a.** Cree una clase base llamada `Punto` que consiste de una coordenada `x` y una coordenada `y`. A partir de esta clase derive una clase llamada `Circulo` que tenga un miembro de datos adicional llamado radio. Para esta clase derivada los miembros `x` y `y` representan las coordenadas del centro de un círculo. Los miembros de la función de la primera clase deberán consistir de un constructor, una función de área llamada `area` que devuelva cero y una función de distancia que devuelva la distancia entre dos puntos, donde

$$\text{distancia} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

   Además, la clase derivada deberá tener un constructor y una función que lo anule llamada area que devuelva el área de un círculo.

b. Incluya las clases construidas para el ejercicio 5a en un programa funcional en C++. Haga que su programa llame a todas las funciones miembro en cada clase. Además llame a la función distancia de la clase base con dos objetos `Circulo` y explique el resultado devuelto por la función.

6. a. Usando las clases construidas para el ejercicio 5a, derive una clase llamada `Cilindro` a partir de la clase derivada `Circulo`. La clase `Cilindro` deberá tener un constructor y una función miembro llamada **area** que determine el área de la superficie del cilindro. Para esta función use el algoritmo *área de superficie* = $2 \pi r (l + r)$, donde r es el radio del cilindro y l es el largo.

b. Incluya las clases construidas para el ejercicio 6a en un programa funcional en C++. Haga que su programa llame a todas las funciones miembro en la clase `Cilindro`.

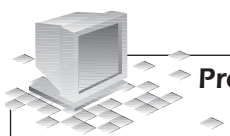
c. ¿Cuál piensa que podría ser el resultado si la función `distancia()` de la clase base fuera llamada con dos objetos `Cilindro`?

7. a. Cree una clase base llamada `Rectangulo` que contenga miembros de datos de largo y ancho. A partir de esta clase derive una clase llamada `Caja` que tenga un miembro de datos adicional llamado profundidad. Los miembros de función de la clase base `Rectangulo` deberá consistir de un constructor y una función de área. La clase derivada `Caja` deberá tener un constructor, una función de volumen y una función de anulación llamada `area` que devuelva el área de superficie de la caja.

b. Incluya las clases construidas para el ejercicio 7a en un programa funcional en C++. Haga que su programa llame a todas las funciones miembro en cada clase y explique el resultado cuando es llamada la función `distancia()` usando un objeto `Caja`.

10.7 POLIMORFISMO

La anulación de una función miembro base usando una función miembro derivada sobrecargada, como se ilustró con la función `calcval()` en el programa 10.13, es un ejemplo de polimorfismo. El **polimorfismo** permite que el mismo nombre de función invoque una respuesta en objetos de una clase base y otra respuesta en objetos de una clase derivada. En algunos casos, sin embargo, este método de anulación no funciona como podría desearse. Para entender por qué sucede esto, considérese el programa 10.14.



Programa 10.14

```
#include <iostream>
#include <cmath>
using namespace std;
```

(Continúa)

(Continuación)

```
// declaración de clase para la clase base

class Uno
{
  protected:
    double a;
  public:
    Uno(double = 2.0);   // constructor
    double f1(double);   // una función miembro
    double f2(double);   // otra función miembro
};

// implementación de clase para Uno

Uno::Uno(double val) // constructor
{
  a = val;
}

double Uno::f1(double num)   // una función miembro
{
  return(num/2);
}
double Uno::f2(double num)   // otra función miembro
{
  return( pow(f1(num),2) );  // eleva al cuadrado el resultado de f1()
}

// declaración de clase para la clase derivada

class Dos : public Uno
{
  public:
    double f1(double);   // esto anula a f1() de la clase Uno
};

// implementación de clase para Dos

double Dos::f1(double num)
{
  return(num/3);
}
```

(Continúa)

(Continuación)

```
int main()
{
  Uno objeto_1; // objeto_1 es un objeto de la clase base
  Dos objeto_2; // objeto_2 es un objeto de la clase derivada

    // llama a f2() usando una llamada al objeto de la clase base
  cout << "El valor calculado usando una llamada a un objeto de clase base es "
       << objeto_1.f2(12) << endl;

    // llama a f2() usando una llamada al objeto de la clase derivada
  cout << "El valor calculado usando una llamada a un objeto de la clase
          derivada es "

       << objeto_2.f2(12) << endl;

  return 0;
}
```

La salida producida por este programa es

```
El valor calculado usando una llamada a un objeto de clase
base es 36
El valor calculado usando una llamada a un objeto de la
clase derivada es 36
```

Como muestra esta salida, se obtiene el mismo resultado sin importar cuál tipo de objeto llame a la función `f2()`. Se produce este resultado porque la clase derivada no tiene una anulación para la función `f2()` de la clase base. Por tanto, ambas llamadas a `f2()` producen que se llame la función `f2()` en la clase base.

Una vez invocada, la función `f2()` de la clase base siempre llamará a la versión de la clase base de `f1()` en lugar de la versión anulada de la clase derivada. Este comportamiento se debe a un proceso conocido como **vínculo de la función**. En llamadas a función normales se usa un vínculo estático. En el **vínculo estático** la determinación de cuál función es llamada se hace en tiempo de compilador. Por tanto, cuando el compilador encuentra por primera vez la función `f1()` en la clase base toma la determinación de que siempre que se llame `f2()`, ya sea desde un objeto de clase base o derivada, en lo subsiguiente llamará a la función `f1()` de la clase base.

En lugar del vínculo estático nos gustaría un método de vinculación que sea capaz de determinar cuál función debería invocarse en tiempo de ejecución, basado en el tipo de objeto que hace la llamada. Este tipo de vinculación se conoce como **vínculo dinámico**. Para lograr el vínculo dinámico C++ proporciona funciones virtuales.

Una especificación de **función virtual** le dice al compilador que cree un apuntador para una función, pero no llena el valor del apuntador hasta que la función se llama en realidad. Luego, en tiempo de ejecución, y basado en el objeto que hace la *llamada, se usa la dirección de la función apropiada*. Crear una función **virtual** es fácil en extremo; todo lo que se requiere es que se coloque la palabra clave **virtual** antes del tipo a devolver de la función en la sección de declaración. Por ejemplo, considérese el programa 10.15, el cual es idéntico al programa 10.14 excepto por la declaración virtual de la función `f1()`.

Programa 10.15

```
#include <iostream>
#include <cmath>
using namespace std;

// declaración de clase para la clase base

class Uno
{
  protected:
    double a;

  public:
    Uno(double = 2.0);   // constructor
    virtual double f1(double);   // una función miembro
    double f2(double);   // otra función miembro
};

// implementación de clase para Uno

Uno::Uno(double val)   // constructor
{
  a = val;
}

double Uno::f1(double num)  // una función miembro
{
  return(num/2);
}

double Uno::f2(double num)  // otra función miembro
{
  return( pow(f1(num),2) ); // eleva al cuadrado el resultado de f1()
}

// declaración de clase para la clase derivada

class Dos : public Uno
{
  public:
    virtual double f1(double);   // esto anula a f1() de la clase Uno
};

// implementación de clase para Dos
```

(Continúa)

(Continuación)

```
double Dos::f1(double num)
{
  return(num/3);
}

int main()
{
  Uno objeto_1; // objeto_1 es un objeto de la clase base
  Dos objeto_2; // objeto_2 es un objeto de la clase derivada

    // llama a f2() usando una llamada al objeto de la clase base
  cout << "El valor calculado usando una llamada a un objeto de clase base es "
       << objeto_1.f2(12) << endl;

    // llama a f2() usando una llamada al objeto de la clase derivada
  cout << "El valor calculado usando una llamada a un objeto de la clase
          derivada es "
       << objeto_2.f2(12) << endl;

  return 0;
}
```

La salida producida por el programa 10.15 es

```
El valor calculado usando una llamada a un objeto de la
clase base es 36
El valor calculado usando una llamada a un objeto de la
clase derivada es 16
```

Como lo muestra esta salida, ahora la función `f2()` llama a versiones diferentes de la función `f1()` sobrecargada basada en el tipo de objeto que hace la llamada. Esta selección, basada en el objeto que hace la llamada, es la definición clásica del comportamiento de una función polimórfica y es causada por el vínculo dinámico impuesto en `f1()` dado que es una función virtual.

Una vez que una función es declarada como virtual *permanece virtual para la siguiente clase derivada con o sin una declaración virtual en la clase derivada*. Por tanto, la segunda declaración virtual en la clase derivada no es estrictamente necesaria, pero debe incluirse tanto por claridad como para asegurar que cualesquiera clases derivadas subsiguientes hereden en forma correcta la función. Considérese el diagrama de herencia ilustrado en la figura 10.9, donde la clase `C` se deriva de la clase `B` y la clase `B` se deriva de la clase `A`.[9] En esta situación, si la función `f1()` es virtual en la clase `A`, pero no es declarada en la clase `B`, no será virtual en la clase `C`. El único requisito adicional es que, una vez que una función ha sido declarada como virtual, el tipo que devuelve y la lista de parámetros de todas las versiones de anulación subsiguientes de las clases derivadas *deben* ser las mismas.

[9]Por convención, como se señaló antes en la sección 10.6, las flechas siempre apuntan de la clase derivada a la clase base.

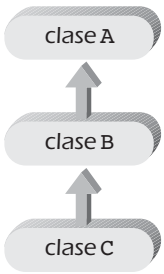


Figura 10.9 *Un diagrama de herencia.*

Ejercicios 10.7

1. Introduzca y ejecute los programas 10.14 y 10.15 en su computadora de modo que entienda la relación entre las llamadas a función en cada programa.
2. Describa los dos métodos que proporciona C++ para implementar el polimorfismo.
3. Describa la diferencia entre vínculo estático y vínculo dinámico.
4. Describa la diferencia entre una función virtual y una función no virtual.
5. Describa qué es polimorfismo y proporcione un ejemplo de comportamiento polimórfico.
6. Discuta, con razones, si el operador de multiplicación proporcionado para los tipos integrados en número entero y de precisión doble es un ejemplo de sobrecarga o de polimorfismo.

10.8 ERRORES COMUNES DE PROGRAMACIÓN

1. Usar un operador de asignación definido por el usuario en una expresión de asignación múltiple cuando el operador no ha sido definido para devolver un objeto.
2. Usar la palabra clave `static` cuando se define un miembro estático de datos o de función. Aquí, la palabra clave `static` deberá utilizarse sólo dentro de la sección de declaración de la clase.
3. Usar la palabra clave `friend` cuando se define una función amiga. La palabra clave `friend` sólo deberá usarse dentro de la sección de declaración de la clase.
4. No instanciar miembros de datos estáticos antes de crear objetos de clase que deben tener acceso a estos miembros de datos.
5. Intentar redefinir el significado de un operador en la forma en que se aplica a los tipos de datos integrados de C++.
6. Redefinir un operador sobrecargado para ejecutar una función no indicada por su significado convencional. Aunque esto funcionará, es un ejemplo de una práctica de programación mala en extremo.

7. Intentar hacer amiga una función operador de conversión, en lugar de hacerla una función miembro.
8. Intentar especificar un tipo de devolución para una función operador de conversión miembro.
9. Intentar anular una función virtual sin usar el mismo tipo y número de argumentos que la función original.
10. Usar la palabra clave `virtual` en la sección de implementación de la clase. Las funciones sólo se declaran como `virtual` en la sección de declaración de la clase.

10.9 RESUMEN DEL CAPÍTULO

1. Puede declararse un operador de asignación para una clase con el prototipo de función

```
void operator=(nombreClase&);
```

Aquí, el argumento es una referencia al nombre de la clase. El tipo devuelto de void impide usar este operador en expresiones de asignación múltiples como `a = b = c`.

2. Un tipo de inicialización que se parece mucho a la asignación ocurre en C++ cuando un objeto es inicializado usando otro objeto de la misma clase. El constructor que ejecuta este tipo de inicialización se llama *constructor de copia* y tiene el prototipo de función

```
nombreClase(const nombreClase& );
```

Esto se representa con frecuencia utilizando la notación *X(X&)*.

3. Cada clase tiene un alcance de clase asociado, el cual se define por el par de llaves, `{}`, que contienen la declaración de la clase. Los miembros de datos y de función son locales para el alcance de su clase y sólo pueden ser usados por objetos declarados para la clase. Si se vuelve a usar un nombre de variable global dentro de una clase, la variable global es ocultada por la variable de clase. Dentro del alcance de la variable de clase puede tenerse acceso a la variable global usando el operador de resolución de alcance, `::`.

4. Para cada objeto de clase se reserva un conjunto separado de ubicaciones de memoria para todos los miembros de datos, excepto aquellos declarados como `static`. Un miembro de datos estático es aquel compartido por todos los objetos de la clase y proporciona un medio de comunicación entre objetos. Los miembros de datos estáticos deben ser declarados como tales dentro de la sección de declaración de la clase y se definen fuera de la sección de declaración.

5. Los miembros de la función `static` se aplican a la clase en su conjunto, más que a objetos individuales. Como tal, un miembro de función `static` sólo puede tener acceso a miembros de datos `static` y a otros miembros de función `static`. Los miembros de función `static` deben ser declarados como tales dentro de la sección de declaración de la clase y se definen fuera de la sección de declaración.

6. Una función que no es miembro puede tener acceso a los miembros de datos `private` de una clase si se le concede categoría de amiga por parte de la clase. Esto se logra declarando la función como amiga dentro de la sección de declaración de la clase. Por tanto, siempre es la clase la que determina cuáles funciones que no son miembro son amigas; una función nunca puede conferirse categoría de amiga a sí misma.

7. Pueden construirse operadores definidos por el usuario para las clases usando funciones operador miembro. Una función operador tiene la forma *operador-<símbolo>*, donde <símbolo> es uno de los siguientes:

```
()  []  ->  new  delete  ++  --  !  ~  *  /  %  +  -
<<  >>  <  <=  >  >=  ++ !=  && ||  &  ^  |  =  +=
-=  *=  /=  %=  &=  ^=  |=  <<= >>=  ,
```

   Por ejemplo, el prototipo de la función `Fecha operator+(int);` declara que el operador de adición se definirá para aceptar un número entero y devolver un objeto `Fecha`.

8. Los operadores definidos por el usuario pueden llamarse en cualquiera de dos formas: como una función convencional con argumentos o como una expresión operador. Por ejemplo, para un operador que tenga la línea de encabezado

```
Fecha Fecha::operator+(int)
```

   si `dte` es un objeto del tipo `Fecha`, las siguientes dos llamadas producen el mismo efecto:

```
dte.operator+(284)
dte + 284
```

9. Las funciones operador también pueden escribirse como funciones amigas. La versión amiga equivalente de una función operador miembro siempre contendrá una referencia adicional a la clase que no es requerida por la función miembro.

10. Hay cuatro categorías de conversiones de tipo de datos. Son conversiones de

    - Tipos integrados a tipos integrados
    - Tipos integrados a tipos definidos por el usuario (clase)
    - Tipos definidos por el usuario (clase) a tipos integrados
    - Tipos definidos por el usuario (clase) a tipos definidos por el usuario (clase)

    Las conversiones de tipo integrado a tipo integrado se hacen usando las reglas de conversión implícitas de C++ o de manera explícita usando moldes. Las conversiones de tipo integrado a tipo definido por el usuario se hacen usando constructores de conversión de tipo. Las conversiones de tipos definidos por el usuario a tipos integrados o a otro tipo definido por el usuario se hacen usando funciones operador de conversión.

11. Un **constructor de conversión de tipo** es un constructor cuyo primer argumento no es un miembro de su clase y cuyos argumentos restantes, si los hay, tienen valores por omisión.

12. Una **función operador de conversión es una función** operador miembro que tiene el nombre de un tipo de datos integrado o de una clase. No tiene argumentos explícitos ni tipo a devolver; más bien, el tipo a devolver es el nombre de la función.

13. La **herencia** es la capacidad para derivar una clase a partir de otra clase. La clase inicial usada como la base para la clase derivada se conoce como base, padre o superclase. La clase derivada se conoce como derivada, hija o subclase.
14. Las funciones de la clase base pueden ser anuladas por funciones de clase derivadas con el mismo nombre. La función de anulación tan sólo es una versión sobrecargada de la función miembro base definida en la clase derivada.
15. El **polimorfismo** es la capacidad de hacer que el mismo nombre de función invoque diferentes respuestas con base en el objeto que haga la llamada a la función. Puede conseguirse usando funciones de anulación o funciones virtuales.
16. En el **vínculo estático** la determinación de cuál función es invocada se hace en tiempo de compilador. En el **vínculo dinámico** la determinación se hace en tiempo de ejecución.
17. Una especificación de **función virtual** designa que deberá tener lugar el vínculo dinámico. La especificación se hace en el prototipo de la función al colocar la palabra clave `virtual` antes del tipo a devolver de la función. Una vez que se ha declarado una función como `virtual` permanece así para todas las clases derivadas en tanto haya un rastro continuo de declaraciones de función a través de la cadena derivada de clases.

Parte tres

Estructuras de datos

Capítulos

11 Arreglos

12 Apuntadores

13 Estructuras

CAPÍTULO 11
Arreglos

TEMAS

11.1 ARREGLOS UNIDIMENSIONALES
ENTRADA Y SALIDA DE VALORES DE ARREGLO

11.2 INICIALIZACIÓN DE ARREGLOS

11.3 DECLARACIÓN Y PROCESAMIENTO DE ARREGLOS BIDIMENSIONALES
ARREGLOS DIMENSIONALES MAYORES

11.4 APLICACIONES
APLICACIÓN 1: ANÁLISIS ESTADÍSTICO
APLICACIÓN 2: MANTENIMIENTO DE UNA LISTA

11.5 ARREGLOS COMO ARGUMENTOS

11.6 LA CLASE DE VECTOR STL

11.7 ERRORES COMUNES DE PROGRAMACIÓN

11.8 RESUMEN DEL CAPÍTULO

11.9 APÉNDICE DEL CAPÍTULO: BÚSQUEDA Y ORDENAMIENTO
ALGORITMOS DE BÚSQUEDA
LA NOTACIÓN DE LA O GRANDE
ALGORITMOS DE ORDENAMIENTO

Todas las variables que se han usado hasta ahora han tenido una característica común: cada variable sólo podía utilizarse para almacenar un solo valor a la vez. Por ejemplo, aunque las variables `clave`*, cuenta y calificacion declaradas en las instrucciones*

```
char clave;
int cuenta;
double calificacion;
```

son todas de tipos de datos diferentes, cada variable sólo puede almacenar un valor del tipo de datos declarado. Estos tipos de variables se llaman ***variables atómicas****. Una variable atómica, la cual también se conoce como* ***variable escalar****, es una variable cuyo valor no puede subdividirse o separarse más en un tipo de datos legítimo.*

Con frecuencia se puede tener un conjunto de valores, todos del mismo tipo de datos, que forman un grupo lógico. Por ejemplo, la figura 11.1 ilustra tres grupos de elementos. El primer grupo es una lista de cinco temperaturas en número de precisión doble, el segundo grupo es una lista de cuatro códigos de caracteres y el último grupo es una lista de seis voltajes en número entero.

Temperaturas	Códigos	Voltajes
95.75	Z	98
83.0	C	87
97.625	K	92
72.5	L	79
86.25		85
		72

Figura 11.1 Tres listas de elementos.

Una lista simple que contiene elementos individuales del mismo tipo de datos se llama arreglo unidimensional. En este capítulo se describe cómo se declaran, inicializan, almacenan dentro de una computadora y usan los arreglos unidimensionales. Además, se explorará el uso de arreglos unidimensionales con ejemplos de programas y se presentan los procedimientos para declarar y usar arreglos multidimensionales.

11.1 ARREGLOS UNIDIMENSIONALES

Un **arreglo unidimensional**, el cual también se conoce como **arreglo de dimensión única**, es una lista de valores relacionados con el mismo tipo de datos que se almacena usando un nombre de grupo único.[1] En C++, como en otros lenguajes de computadora, el nombre del grupo se conoce como el **nombre del arreglo**. Por ejemplo, considérese la lista de temperaturas ilustrada en la figura 11.2.

Temperaturas
95.75
83.0
97.625
72.5
86.25

Figura 11.2 Una lista de temperaturas.

[1]Hay que observar que las listas pueden implementarse en una variedad de formas. Un arreglo tan sólo es una implementación de una lista en la que todos los elementos de la lista son del mismo tipo y cada elemento es almacenado de manera consecutiva en un conjunto de ubicaciones de memoria contiguas.

Todas las temperaturas en la lista son números en punto flotante y deben declararse como tales. Sin embargo, los elementos individuales en la lista no tienen que declararse por separado. Los elementos en la lista pueden ser declarados como una sola unidad y almacenarse bajo un nombre de variable común llamado nombre del arreglo. Por comodidad, se elegirá `temp` como el nombre para la lista mostrada en la figura 11.2. Especificar que `temp` va a almacenar cinco valores en punto flotante individuales requiere la instrucción de declaración `double temp[5]`. Hay que observar que esta instrucción de declaración proporciona el nombre del arreglo (o lista), el tipo de datos de los elementos en el arreglo y el número de elementos en el arreglo. Es un ejemplo específico de la instrucción de declaración de arreglo general que tiene la sintaxis:

tipo-de-datos nombreArreglo[número-de-elementos]

La buena práctica de programación requiere definir el número de elementos en el arreglo como una constante antes de declarar el arreglo. Por tanto, la declaración del arreglo previa para `temp` se declararía, en la práctica, usando dos elementos, como:

```
const int NUMELS = 5;  // define una constante para el número de
                          elementos
double temp[NUMELS];   // declara el arreglo
```

Otros ejemplos de declaraciones de arreglo usando esta sintaxis de dos líneas son:

```
const int NUMELS = 6;
int voltios[NUMELS];

const int TAMARREGLO = 4;
char codigo[TAMARREGLO];

const int TAMANHO = 100;
double cantidad[TAMANHO];
```

En estas instrucciones de declaración, a cada arreglo se le asigna suficiente memoria para contener el número de elementos de datos dado en la instrucción de declaración. Por tanto, el arreglo nombrado `voltios` tiene almacenamiento reservado para seis números enteros, el arreglo llamado `codigo` tiene almacenamiento reservado para cuatro caracteres y el arreglo llamado `cantidad` tiene almacenamiento reservado para 100 números de precisión doble. Los identificadores de la constante, `NUMELS, TAMARREGLO` y `TAMANHO` son nombres seleccionados por el programador.

La figura 11.3 ilustra el almacenamiento reservado para los arreglos `voltios` y `codigo`.

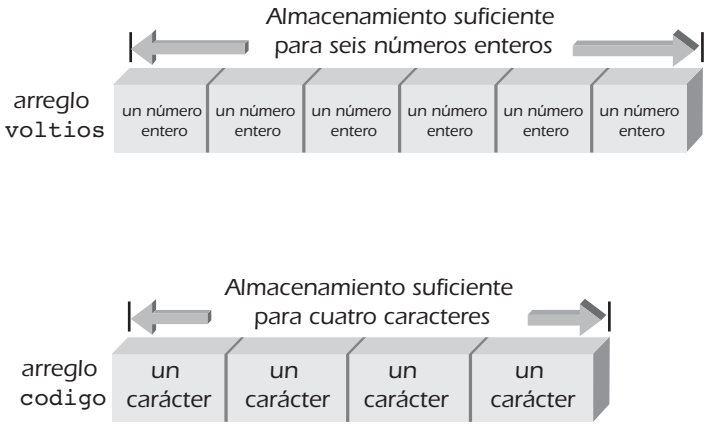


Figura 11.3 *Los arreglos* `voltios` *y* `codigo` *en la memoria.*

Cada elemento en un arreglo se llama **elemento** o **componente** del arreglo. Los elementos individuales almacenados en los arreglos ilustrados en la figura 11.3 se almacenan de manera secuencial, con el primer elemento del arreglo almacenado en la primera ubicación reservada, el segundo elemento almacenado en la segunda ubicación reservada, y así en forma sucesiva hasta que el último elemento es almacenado en la última ubicación reservada. Esta asignación de almacenamiento contiguo para la lista es una característica clave de los arreglos porque proporciona un mecanismo simple para localizar con facilidad cualquier elemento individual en la lista.

Dado que los elementos en el arreglo se almacenan de manera secuencial, puede tenerse acceso a cualquier elemento individual dando el nombre del arreglo y la posición del elemento. Esta posición se llama valor ***índice*** o ***subíndice*** del elemento (los dos términos son sinónimos). Para un arreglo unidimensional, el primer elemento tiene un índice de 0, el segundo elemento tiene un índice de 1, etc. En C++, el nombre del arreglo y el índice del elemento deseado se combinan enlistando el índice entre corchetes después del nombre del arreglo. Por ejemplo, dada la declaración `double temp[5]`.

`temp[0]` se refiere a la primera temperatura almacenada en el arreglo `temp`

`temp[1]` se refiere a la segunda temperatura almacenada en el arreglo `temp`

`temp[2]` se refiere a la tercera temperatura almacenada en el arreglo `temp`

`temp[3]` se refiere a la cuarta temperatura almacenada en el arreglo `temp`

`temp[4]` se refiere a la quinta temperatura almacenada en el arreglo `temp`

La figura 11.4 ilustra el arreglo `temp` en la memoria con la designación correcta para cada elemento del arreglo. Se dice que cada elemento individual es una **variable indexada** o una **variable subindexada**, ya que deben usarse tanto un nombre de variable como un valor índice o subíndice para hacer referencia al elemento. Recuérdese que el valor índice o de subíndice da *la posición* del elemento en el arreglo.

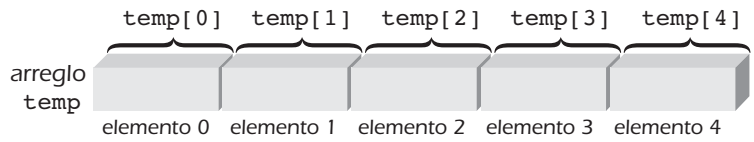


Figura 11.4 *Identificación de elementos individuales del arreglo.*

La variable subindexada, `temp[0]`, se lee como `"temp subíndice cero"`. Ésta es una forma abreviada de decir `"el arreglo temp con subíndice cero"`. Del mismo modo, `temp[1]` se lee como `"temp subíndice uno"`, `temp[2]` como `"temp subíndice dos"`, etcétera.

Aunque puede parecer inusual hacer referencia al primer elemento con un índice de cero, hacerlo así incrementa la velocidad de la computadora cuando tiene acceso a los elementos del arreglo. De manera interna, invisible para el programador, la computadora usa el índice como una compensación de la posición inicial del arreglo. Como se ilustra en la figura 11.5, el índice le indica a la computadora cuántos elementos saltar, empezando desde el principio del arreglo, para obtener el elemento deseado.

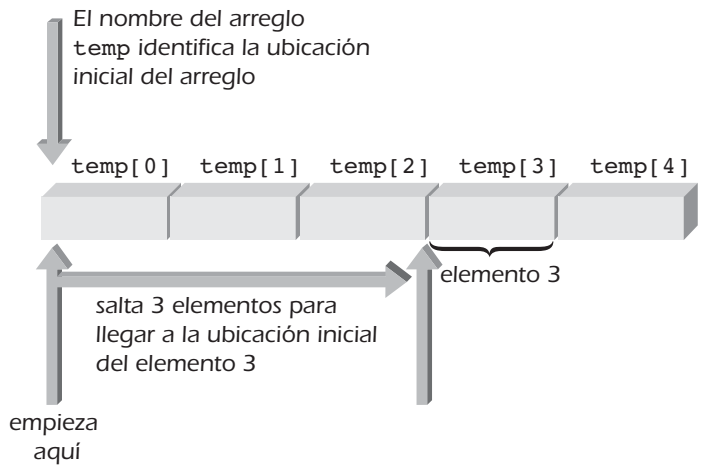


Figura 11.5 *Acceso a un elemento individual del arreglo: el elemento 3.*

Las variables subindexadas pueden usarse en cualquier parte en que sean válidas las variables escalares. Son ejemplos que utilizan los elementos del arreglo temp:

```
temp[0] = 95.75;
temp[1] = temp[0] - 11.0;
temp[2] = 5.0 * temp[0];
temp[3] = 79.0;
temp[4] = (temp[1] + temp[2] - 3.1) / 2.2;
suma = temp[0] + temp[1] + temp[2] + temp[3] + temp[4];
```

El subíndice contenido dentro de los corchetes no necesita ser una constante en número entero; cualquier expresión que evalúe a un número entero puede usarse como un subíndice.[2] En cada caso, por supuesto, el valor de la expresión debe estar dentro del rango de subíndices válidos definidos cuando se declaró el arreglo. Por ejemplo, suponiendo que `i` y `j` son variables `int`, las siguientes variables subindexadas son válidas:

```
temp[i]
temp[2*i]
temp[j-i]
```

Una ventaja importante en extremo de usar expresiones en número entero como subíndices es que permite secuenciar a través de un arreglo usando un ciclo. Esto hace innecesarias instrucciones como

```
suma = temp[0] + temp[1] + temp[2] + temp[3] + temp[4];
```

Los valores de subíndice en esta instrucción pueden reemplazarse por un contador de ciclo `for` para tener acceso a cada elemento en el arreglo en forma secuencial. Por ejemplo, el código

```
suma = 0;                  // inicializa la suma en cero
for (i = 0; i < 5; i++)
  suma = suma + temp[i];   // agrega un valor
```

recupera de manera secuencial cada elemento del arreglo y agrega el elemento a `suma`. Aquí la variable `i` se usa como el contador en el ciclo `for` y como un subíndice. Conforme `i` se incrementa en uno cada vez a través del ciclo, el siguiente elemento en el arreglo es referenciado. El procedimiento para agregar los elementos del arreglo dentro del ciclo `for` es similar al procedimiento de acumulación que se ha usado muchas veces antes.

La ventaja de usar un ciclo `for` para procesar en secuencia a través de un arreglo se hace evidente cuando se trabaja con arreglos grandes. Por ejemplo, si el arreglo `temp` contuviera 100 valores en lugar de sólo cinco, cambiar el número 5 a 100 en la instrucción `for` es suficiente para procesar en secuencia los 100 elementos y agregar cada temperatura a la suma.

Como otro ejemplo del uso de un ciclo `for` para secuenciar un arreglo, suponga que se desea localizar el valor máximo en un arreglo de 1000 elementos llamado `voltios`. El procedimiento que se usará para localizar el valor máximo es suponer inicialmente que el primer elemento en el arreglo es el número más grande. Luego, conforme avanzamos en secuencia a través del arreglo, el máximo se compara con cada elemento. Cuando se localiza un elemento con un valor mayor, ese elemento se convierte en el nuevo máximo. El siguiente código hace el trabajo.

```
const int NUMELS = 1000;

maximo = voltios[0];       // establece el máximo en el elemento cero
for (i = 1; i < NUMELS; i++) // ciclo a traves del resto del arreglo
  if (voltios[i] > maximo)   // compara cada elemento con el máximo
    maximo = voltios[i];     // captura el nuevo valor alto
```

[2]Nota: Algunos compiladores permiten variables de punto flotante como subíndices; en estos casos el valor en punto flotante es truncado a un valor entero.

En este código la instrucción `for` consiste en una instrucción `if`. La búsqueda de un nuevo valor máximo comienza con el elemento 1 del arreglo y continúa hasta el último elemento. Cada elemento es comparado con el máximo actual, y cuando se encuentra un valor más alto éste se convierte en el nuevo máximo.

Entrada y salida de valores de arreglo

A los elementos individuales del arreglo se les pueden asignar valores de manera interactiva usando un objeto de corriente `cin`. Son ejemplos de instrucciones de introducción de datos individuales:

```
cin >> temp[0];
cin >> temp[1] >> temp[2] >> temp[3];
cin >> temp[4] >> voltios[6];
```

En la primera instrucción se leerá y almacenará un solo valor en la variable nombrada `temp[0]`. La segunda instrucción causará que se lean y almacenen tres valores en las variables `temp[1], temp[2]` y `temp[3]`, respectivamente. Por último, puede usarse la última instrucción `cin` para leer valores en las variables `temp[4]` y `voltios[6]`.

De manera alternativa, puede utilizarse un ciclo `for` para recorrer en forma cíclica el arreglo para la introducción interactiva de datos. Por ejemplo, el código

```
const int NUMELS = 5;

for (i = 0; i < NUMELS; i++)
{
  cout << "Introduzca una temperatura: ";
  cin  >> temp[i];
}
```

insta al usuario a introducir cinco temperaturas. La primera temperatura introducida se almacena en `temp[0]`, la segunda temperatura introducida se almacena en `temp[1]`, y así de manera sucesiva hasta que se han introducido cinco temperaturas.

Hay que hacer una advertencia acerca de almacenar datos en un arreglo. C++ no comprueba el valor del índice que se está usando (llamado ***verificación de límites***). Si se ha declarado que un arreglo consta de 10 elementos, por ejemplo, y se usa un índice de 12, el cual está fuera de los límites del arreglo, C++ no notificará el error cuando se compile el programa. El programa intentará tener acceso al elemento 12 saltándose el número apropiado de bytes desde el inicio del arreglo. Por lo general esto produce una caída del programa, pero no siempre. Si la ubicación referenciada en sí contiene un valor del tipo de datos correcto, el nuevo valor tan sólo sobrescribirá el valor en las ubicaciones de memoria referenciadas. Esto conduce a más errores, los cuales son difíciles de localizar, en particular cuando la variable asignada de manera legítima a la ubicación de almacenamiento se usa en un punto diferente del programa.

Durante la salida, pueden desplegarse elementos individuales del arreglo usando el objeto `cout` o secciones completas del arreglo incluyendo una instrucción `cout` dentro de un ciclo `for`. Son ejemplos de esto

```
cout << voltios[6];
```

y

```
cout << "El valor del elemento " << i << " es " << temp[i];
```

Punto de información

Tipos de datos agregados

En contraste con los tipos atómicos, como datos en número entero y de punto flotante, hay tipos agregados. Un tipo agregado, el cual también se conoce como *tipo estructurado* y estructura de datos, es cualquier tipo cuyos valores puedan descomponerse y están relacionados por alguna estructura definida. Además, debe haber operaciones disponibles para recuperar y actualizar valores individuales en la estructura de datos.

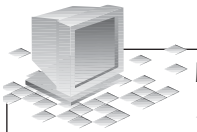
Los arreglos unidimensionales son ejemplos de un tipo estructurado. En un arreglo unidimensional, como un arreglo de números enteros, el arreglo está compuesto por valores enteros individuales, donde los números enteros están relacionados por su posición en la lista. Las variables indexadas proporcionan los medios para tener acceso a los valores en el arreglo y modificarlos.

y

```
const int NUMELS = 20;

for (k = 5; k < NUMELS; k++)
   cout << k << "   " << cantidad[k] << endl;
```

La primera instrucción despliega el valor de la variable subindexada `voltios[6]`. La segunda instrucción despliega el valor del subíndice `i` y el valor de `temp[i]`. Antes que pueda ejecutarse esta instrucción, `i` tendría que tener un valor asignado. Por último, el ejemplo final incluye un objeto `cout` dentro de un ciclo `for`. Se despliegan tanto el valor del índice como el valor de los elementos de 5 a 20.

El programa 11.1 ilustra estas técnicas de entrada y salida usando un arreglo llamado `temp` que se define para almacenar cinco números enteros. Se incluyen en el programa dos ciclos `for`. El primer ciclo `for` se usa para pasar por cada elemento del arreglo y permite al usuario introducir valores individuales del arreglo. Después que se han introducido cinco valores, se usa el segundo ciclo `for` para desplegar los valores almacenados.



Program 11.1

```
#include <iostream>
using namespace std;

int main()
{
  const int TEMPMAX = 5;
  int i, temp[TEMPMAX];

  for (i = 0; i < TEMPMAX; i++)      // Se introducen las temperaturas
  {
    cout << "Introduzca una temperatura: ";
    cin  >> temp[i];
  }

  cout << endl;

  for (i = 0; i << TEMPMAX; i++)      // Se imprimen las temperaturas
    cout << "temperatura " << i << " es " << temp[i] << endl;

  return 0;
}
```

A continuación se presenta una muestra de la ejecución del programa 11.1:

```
Introduzca una temperatura: 85
Introduzca una temperatura: 90
Introduzca una temperatura: 78
Introduzca una temperatura: 75
Introduzca una temperatura: 92

temperatura 0 es 85
temperatura 1 es 90
temperatura 2 es 78
temperatura 3 es 75
temperatura 4 es 92
```

Al revisar la salida producida por el programa 11.1, debe ponerse particular atención a la diferencia entre el valor del índice desplegado y el valor numérico almacenado en el correspondiente elemento del arreglo. El valor del índice se refiere a la ubicación del elemento en el arreglo, mientras la variable subindexada se refiere al valor almacenado en la ubicación designada.

Además de tan sólo desplegar los valores almacenados en cada elemento del arreglo, los elementos también pueden procesarse al referenciar de manera apropiada el elemento deseado. Por ejemplo, en el programa 11.2, el valor de cada elemento se acumula en un total, el cual se despliega al completar el despliegue individual de cada elemento del arreglo.

Programa 11.2

```
#include <iostream>
using namespace std;

int main()
{
  const int TEMPMAX = 5;
  int i, temp[TEMPMAX], total = 0;

  for (i = 0; i < TEMPMAX; i++)      // Se introducen las temperaturas
  {
    cout << "Introduzca una temperatura: ";
    cin  >> temp[i];
  }

  cout << "\nEl total de las temperaturas";

  for (i = 0; i < TEMPMAX; i++)     // Despliega y calcula el total de
                                       las temperaturas
  {
    cout << "   " << temp[i];
    total =  total + temp[i];
  }

  cout << " es " << total << endl;

  return 0;
}
```

A continuación se presenta una muestra de la ejecución del programa 11.2:

```
Introduzca una temperatura: 85
Introduzca una temperatura: 90
Introduzca una temperatura: 78
Introduzca una temperatura: 75
Introduzca una temperatura: 92

El total de las temperaturas 85   90   78   75   92   es   420
```

Hay que observar que en el programa 11.2, a diferencia del programa 11.1, sólo se despliegan los valores almacenados en cada elemento del arreglo y no los números índice. Aunque el segundo ciclo `for` se usó para acumular el total de cada elemento, la acumulación también podría haberse logrado en el primer ciclo colocando la instrucción `total = total + temp[i];` después de la instrucción `cin` usada para introducir un valor. También observe que la instrucción `cout` usada para desplegar el total se hace fuera del segundo ciclo `for`, así que el total se despliega sólo una vez, después que se han agregado todos los valores al total. Si esta instrucción `cout` se colocara dentro del ciclo `for` se des-

plegarían cinco totales, con sólo el último total desplegado conteniendo la suma de todos los valores del arreglo.

Ejercicios 11.1

1. Escriba declaraciones de arreglo para lo siguiente:
   a. una lista de 100 voltajes de precisión doble
   b. una lista de 50 temperaturas de precisión doble
   c. una lista de 30 caracteres, cada uno representando un código
   d. una lista de 100 años en número entero
   e. una lista de 32 velocidades de precisión doble
   f. una lista de 1000 distancias de precisión doble
   g. una lista de 6 números de código enteros
2. Escriba una notación apropiada para el primero, tercero y séptimo elementos de los siguientes arreglos:

   a. `int calificaciones[20]`
   b. `double voltios[10]`
   c. `double amperes[16]`
   d. `int dist[15]`
   e. `double velocidad[25]`
   f. `double tiempo[100]`

3. a. Escriba instrucciones de entrada individuales usando `cin` que puedan usarse para introducir valores en el primero, tercero y séptimo elementos de cada uno de los arreglos declarados en los ejercicios 2a a 2f.
   b. Escriba un ciclo `for` que pueda usarse para introducir valores para el arreglo completo declarado en los ejercicios 2a a 2f.
4. a. Escriba instrucciones de salida individuales usando `cout` que puedan utilizarse para imprimir los valores del primero, tercero y séptimo elementos de cada uno de los arreglos declarados en los ejercicios 2a a 2f.
   b. Escriba un ciclo `for` que pueda usarse para desplegar valores para el arreglo completo declarado en los ejercicios 2a a 2f.
5. Enliste los elementos que serán desplegados por las siguientes secciones de código:

   a.
   ```
   for (m = 1; m <= 5; m++)
      cout << a[m] << " ";
   ```
   b.
   ```
   for (k = 1; k <= 5; k = k + 2)
      cout <<  a[k] << " ";
   ```
   c.
   ```
   for (j = 3; j <= 10; j++)
       cout << b[j] << " ";
   ```
   d.
   ```
   for (k = 3; k <= 12; k = k + 3)
       cout << b[k] << " ";
   ```
   e.
   ```
   for (i = 2; i < 11; i = i + 2)
       cout << c[i] << " ";
   ```

6. a. Escriba un programa para introducir los siguientes valores en un arreglo nombrado `voltios`: 11.95, 16.32, 12.15, 8.22, 15.98, 26.22, 13.54, 6.45, 17.59. Después que se hayan introducido los datos, haga que su programa despliegue los valores.

b. Repita el ejercicio 6a, pero después que se hayan introducido los datos, haga que su programa despliegue en la forma siguiente:

```
11.95    16.32    12.15
 8.22    15.98    26.22
13.54     6.45    17.59
```

7. Escriba un programa para introducir ocho números enteros en un arreglo llamado `temp`. Conforme se introduce cada número, sume los números en un total. Después que se hayan introducido todos los números, despliegue los números y su promedio.

8. a. Escriba un programa para introducir 10 números enteros en un arreglo nombrado `fmax` y determine el valor máximo introducido. Su programa deberá contener sólo un ciclo y el máximo deberá ser determinado conforme se introducen los valores de los elementos del arreglo. (*Sugerencia*: Establezca el máximo igual al primer elemento del arreglo, el cual deberá ser introducido antes que se use el ciclo para introducir los valores restantes del arreglo.)

b. Repita el ejercicio 8a, siguiendo la pista tanto del elemento máximo en el arreglo como el número de índice para el máximo. Después de desplegar los números, imprima estos dos mensajes

```
El valor máximo es:
Este es el elemento numero _____ en la lista de numeros
```

Haga que su programa despliegue los valores correctos en lugar del subrayado en los mensajes.

c. Repita el ejercicio 8b, pero haga que su programa localice el mínimo de los datos introducidos.

9. a. Escriba un programa para introducir los siguientes números enteros en un arreglo llamado `calificaciones`: 89, 95, 72, 83, 99, 54, 86, 75, 92, 73, 79, 75, 82, 73. Conforme introduce cada número, sume los números a un total. Después que todos los números son introducidos y se obtiene el total, calcule el promedio de los números y use el promedio para determinar la desviación de cada valor del promedio. Almacene cada desviación en un arreglo llamado `desviacion`. Cada desviación se obtiene como el valor del elemento menos el promedio de todos los datos. Haga que su programa despliegue cada desviación al lado de su elemento correspondiente del arreglo calificaciones.

b. Calcule la varianza de los datos usados en el ejercicio 9a. La varianza se obtiene elevando al cuadrado cada desviación individual y dividiendo la suma de las desviaciones cuadradas entre el número de `desviaciones`.

10. Escriba un programa que especifique tres arreglos unidimensionales llamados `corriente, resistencia` y `voltios`. Cada arreglo deberá ser capaz de contener 10 elementos. Usando un ciclo `for`, introduzca valores para los arreglos `corriente` y `resistencia`. Las entradas en el arreglo voltios deberán ser el producto de los valores correspondientes en los arreglos corriente y `resistencia` (por tanto, `voltios[i] = corriente[i] * resistencia[i]`). Después que se han introducido todos los datos, despliegue la siguiente salida:

```
Corriente        Resistencia        Voltios
```

Bajo cada encabezado de columna despliegue el valor apropiado.

11.2 INICIALIZACIÓN DE ARREGLOS

Los elementos del arreglo pueden inicializarse dentro de sus instrucciones de declaración de la misma manera como lo hacen las variables escalares, excepto que los elementos inicializados deben incluirse entre llaves. Son ejemplos de dicha inicialización

```
int temp[5] = {98, 87, 92, 79, 85};
char codigos[6] = {'m', 'u', 'e', 's', 't', 'r', 'a'};
double pendientes[7] = {11.96, 6.43, 2.58, .86, 5.89,7.56, 8.22];
```

Los inicializadores se aplican en el orden en que se escriben, con el primer valor usado para inicializar el elemento 0, el segundo valor usado para inicializar el elemento 1, etc., hasta que todos los valores se han usado. Por tanto, en la declaración

```
int temp[5] = {98, 87, 92, 79, 85};
```

`temp[0]` es inicializado en 98, `temp[1]` es inicializado en 87, `temp[2]` es inicializado en 92, `temp[3]` es inicializado en 79 y `temp[4]` es inicializado en 85.

Debido a que el espacio en blanco es ignorado en C++, las inicializaciones pueden continuarse a lo largo de múltiples líneas. Por ejemplo, la declaración

```
int galones[20] = {19, 16, 14, 19, 20, 18, // la inicializacion
                                           de valores
                   12, 10, 22, 15, 18, 17,  // puede extenderse a
                                           lo largo
                   16, 14, 23, 19, 15, 18,  // multiples lineas
                   21, 5};
```

usa cuatro líneas para inicializar todos los elementos del arreglo.

Si el número de inicializadores es menor que el número de elementos declarado enlistados entre corchetes, los inicializadores se aplican empezando con el elemento 0 del arreglo. Por tanto, en la declaración

```
double largo[7] = {7.8, 6.4, 4.9, 11.2};
```

sólo se inicializan `largo[0]`, `largo[1]`, `largo[2]` y `largo[3]` con los valores enlistados. Los otros elementos del arreglo serán inicializados en cero.

Por desgracia, no hay un método para indicar la repetición de un valor de inicialización o para inicializar más tarde elementos del arreglo sin especificar primero valores para elementos anteriores.

Una característica única de los inicializadores es que puede omitirse el tamaño de un arreglo cuando los valores de inicialización están incluidos en la instrucción de declaración. Por ejemplo, la declaración

```
int galones[] = {16, 12, 10, 14, 11];
```

reserva suficiente espacio de almacenamiento para cinco elementos. Del mismo modo, las siguientes dos declaraciones son equivalentes:

```
char codigos[6] = {'m', 'u', 'e', 's', 't', 'r', 'a'};
char codigos[] = {'m', 'u', 'e', 's', 't', 'r', 'a'};
```

Ambas declaraciones apartan siete ubicaciones de caracteres para un arreglo llamado `codigos`. También puede usarse una simplificación interesante y útil cuando se inicializan arreglos de caracteres. Por ejemplo, la declaración

```
char codigos[] = "muestra"; // sin llaves ni comas
```

usa la cadena `"muestra"` para inicializar el arreglo codigos. Recuerde que una cadena es cualquier secuencia de caracteres encerrados entre comillas. Esta última declaración crea un arreglo llamado codigos que tiene ocho elementos y llena el arreglo con los ocho caracteres ilustrados en la figura 11.6. Los primeros siete caracteres, como se esperaba, consisten en las letras `m, u, e, s, t, r` y `a`. El último carácter, el cual es la secuencia de escape \0, se llama **carácter nulo**. El carácter nulo se agrega de manera automática a todas las cadenas que se usan para inicializar un arreglo de carácter y es lo que distingue a una cadena C de una cadena de la clase `string`. Este carácter tiene un código de almacenamiento interno que numéricamente es igual a cero (el código de almacenamiento para el carácter cero tiene un valor numérico de 48 decimal, así que no puede confundirlos la computadora), y se usa como un marcador, o centinela, para marcar el final de una cadena.

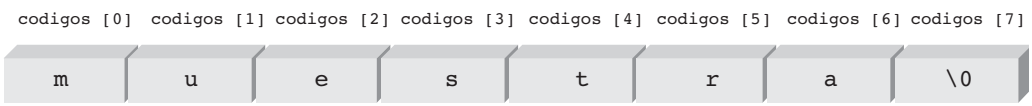
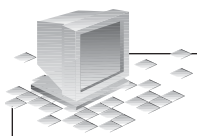


Figura 11.6 *Inicializar un arreglo de carácter con una cadena agrega un carácter \0 de terminación..*

Una vez que se han asignado los valores a los elementos del arreglo, ya sea a través de la inicialización dentro de la instrucción de declaración o usando entrada interactiva, los elementos del arreglo pueden ser procesados como se describió en la sección anterior. Por ejemplo, el programa 11.3 ilustra la inicialización de elementos del arreglo dentro de la declaración del arreglo y luego usa un ciclo `for` para localizar el valor máximo almacenado en el arreglo.



Programa 11.3

```
#include <iostream>
using namespace std;

int main()
{
  const int ELMAX = 5;

  int i, max, nums[ELMAX] = {2, 18, 1, 27, 16};

  max = nums[0];

  for (i = 1; i < ELMAX; i++)
    if (max < nums[i])
      max = nums[i];

  cout << "El valor máximo es " << max << endl;

  return 0;
}
```

La salida producida por el programa 11.3 es

```
El valor maximo es 27
```

Ejercicios 11.2

1. Escriba declaraciones de arreglos, incluyendo inicializadores, para lo siguiente
   - **a.** una lista de 10 voltajes en números enteros: 89, 75, 82, 93, 78, 95, 81, 88, 77, 82
   - **b.** una lista de cinco pendientes en número de precisión doble: 11.62, 13.98, 18.45, 12.68, 14.76
   - **c.** una lista de 100 distancias en número de precisión doble; las primeras seis distancias son 6.29, 6.95, 7.25, 7.35, 7.40, 7.42
   - **d.** una lista de 64 temperaturas en número de precisión doble; las primeras 10 temperaturas son 78.2, 69.6, 68.5, 83.9, 55.4, 67.0, 49.8, 58.3, 62.5, 71.6
   - **e.** una lista de 15 códigos de carácter; los primeros siete códigos son f, j, m, q, t, w, z
2. Escriba una instrucción de declaración de arreglo que almacene los siguientes valores en un arreglo llamado `voltios`: 16.24, 18.98, 23.75, 16.29, 19.54, 14.22, 11.13, 15.39. Incluya estas instrucciones en un programa que despliegue los valores en el arreglo.
3. Escriba un programa que use una instrucción de declaración de arreglo para inicializar los siguientes números en un arreglo llamado `pendientes:` 17.24, 25.63, 5.94, 33.92, 3.71, 32.84, 35.93, 18.24, 6.92. Su programa deberá localizar y desplegar los valores máximo y mínimo en el arreglo.
4. Escriba un programa que almacene los siguientes valores en un arreglo llamado `resistencia`: 16, 27, 39, 56 y 81. Su programa también deberá crear dos arreglos llamados `corriente` y `potencia`, cada uno capaz de almacenar cinco números de precisión doble. Usando un ciclo `for` y una instrucción `cin`, haga que su programa acepte cinco números introducidos por el usuario en el arreglo `corriente` cuando se esté ejecutando el programa. Su programa deberá almacenar el producto de los valores correspondientes del cuadrado del arreglo `corriente` y el arreglo `resistencia` en el arreglo `potencia` (por ejemplo, `potencia[1] = resistencia[1] * pow(corriente[1],2)` y despliegue la siguiente salida (llene la tabla de manera apropiada):

```
Resistencia      Corriente      Potencia
16                  .              .
27                  .              .
39                  .              .
56                  .              .
81                  .              .
                                  --------
Total:                            .
```

5. **a.** Escriba una declaración para almacenar la cadena `"Esta es una prueba"` en un arreglo llamado `pruebacadena`. Incluya la declaración en un programa para desplegar el mensaje usando el siguiente ciclo:

```
for (i = 0; i < DESPLIEGUENUM; i++)
  cout << pruebacadena[i];
```

donde `DESPLIEGUENUM` es una constante nombrada para el número 14.

b. Modifique la instrucción `for` en el ejercicio 5a para desplegar sólo los caracteres p, r, u, e, b y a del arreglo.
c. Incluya la declaración del arreglo escrita en el ejercicio 5a en un programa que use el objeto `cout` para desplegar caracteres en el arreglo. Por ejemplo, la instrucción `cout << pruebacadena;` causará que se despliegue la cadena almacenada en el `arreglo pruebacadena`. Utilizar esta instrucción requiere que el último carácter en el arreglo sea el marcador de fin de cadena `\0`.
d. Repita el ejercicio 5a usando un ciclo `while`. (*Sugerencia*: Detenga el ciclo cuando se detecte la secuencia de escape \0. Puede usarse la expresión `while (pruebacadena[i] != '\0').`)

11.3 DECLARACIÓN Y PROCESAMIENTO DE ARREGLOS BIDIMENSIONALES

Un ***arreglo bidimensional***, el cual a veces se denomina tabla, consiste de filas y columnas de elementos. Por ejemplo, el arreglo de números

8	16	9	52
3	15	27	6
14	25	2	10

se llama arreglo bidimensional de números enteros. Este arreglo consiste de tres filas y cuatro columnas. Para reservar almacenamiento para este arreglo, deben incluirse el número de filas y el número de columnas en la declaración del arreglo. Llamando al arreglo `val`, la especificación correcta para este arreglo bidimensional es

```
int val[3][4];
```

Del mismo modo, las declaraciones

```
double voltios[10][5];
char codigo[6][26];
```

declaran que el arreglo `voltios` consta de 10 filas y 5 columnas de números de precisión doble y que el arreglo `codigo` consta de 6 filas y 26 columnas, con cada elemento capaz de contener un carácter.

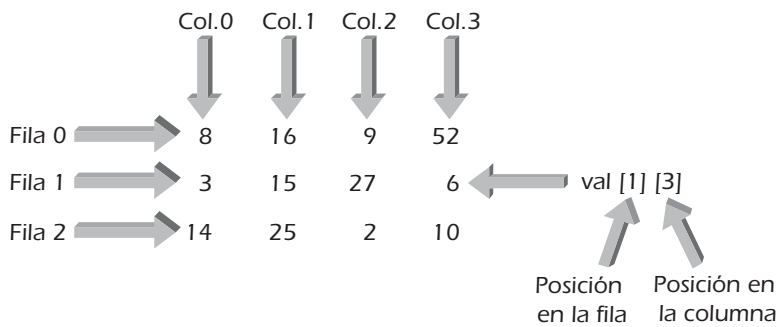


Figura 11.7 Cada elemento del arreglo es identificado por su posición en las filas y columnas.

Para localizar cada elemento en un arreglo bidimensional, un elemento se identifica por su posición en el arreglo. Como se ilustra en la figura 11.7, el término `val[1][3]` identifica en forma única al elemento en la fila 1, columna 3. Como con las variables de arreglo unidimensional, las variables de arreglo bidimensional pueden usarse en cualquier parte en que sean válidas las variables escalares. Son ejemplos del uso de elementos del arreglo `val`

```
watts = val[2][3];
val[0][0] = 62;
nuevonum = 4 * (val[1][0] - 5);
sumaFila0 = val[0][0] + val[0][1] + val[0][2] + val[0][3];
```

La última instrucción causa que los valores de los cuatro elementos en la fila 0 se sumen y la suma se almacene en la variable escalar `sumaFila0.`

Como con los arreglos unidimensionales, los arreglos bidimensionales pueden inicializarse desde dentro de sus instrucciones de declaración. Esto se hace enlistando los valores iniciales dentro de llaves y separándolos con comas. Además, las llaves pueden usarse para separar filas individuales. Por ejemplo, la declaración

```
int val[3][4] = { {8,16,9,52},
                  {3,15,27,6},
                  {14,25,2,10} };
```

declara que `val` es un arreglo de números enteros con tres filas y cuatro columnas, con sus valores iniciales dados en la declaración. El primer conjunto de llaves internas contiene los valores para la fila 0 del arreglo, el segundo conjunto de llaves internas contiene los valores para la fila 1, y el tercer conjunto de llaves, los valores para la fila 2.

Aunque siempre se requieren las comas en las llaves de inicialización, las llaves internas pueden omitirse. Por tanto, la inicialización para `val` puede escribirse como

```
int val[3][4] = {8,16,9,52,
                 3,15,27,6,
                14,25,2,10};
```

La separación de los valores iniciales en filas en la instrucción de declaración no es necesaria en virtud que el compilador asigna valores comenzando con el elemento `[0][0]` y procede fila por fila para llenar los valores restantes. Por tanto, la inicialización

```
int val[3][4] = {8,16,9,52,3,15,27,6,14,25,2,10};
```

es igual de válida pero no ilustra con claridad a otro programador dónde termina una fila y comienza otra.

Como se ilustra en la figura 11.8, la inicialización de un arreglo bidimensional se hace en el orden de las filas. Primero se inicializan los elementos de la primera fila, luego se inicializan los elementos de la segunda fila, y así en forma sucesiva, hasta que se completan las inicializaciones. Este ordenamiento por filas también es el mismo ordenamiento usado para almacenar arreglos bidimensionales. Es decir, el elemento `[0][0]` del arreglo es almacenado primero, seguido por el elemento `[0][1]`, luego el elemento `[0][2]`, etc. Después de los elementos de la primera fila están los elementos de la segunda fila, y así en forma sucesiva para todas las filas en el arreglo.

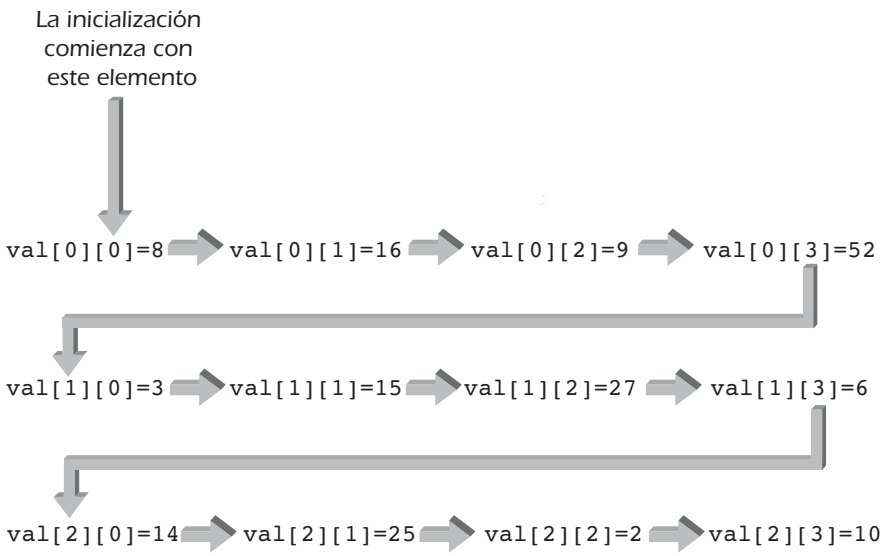


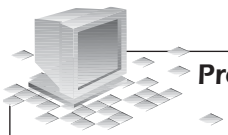
Figura 11.8 *Almacenamiento e inicialización del arreglo* `val`.

Como con los arreglos unidimensionales, los arreglos bidimensionales pueden desplegarse por notación de elemento individual o usando ciclos (ya sea `while` o `for`). Esto se ilustra con el programa 11.4, el cual despliega todos los elementos de un arreglo bidimensional de tres por cuatro usando dos técnicas diferentes. Hay que observar en el programa 11.4 que se han usado constantes para definir las filas y columnas del arreglo.

A continuación se muestra el despliegue producido por el programa 11.4.

```
Despliegue del arreglo val por elemento explícito
   8  16   9  52
   3  15  27   6
  14  25   2  10

Despliegue del arreglo val usando un ciclo for anidado
   8  16   9  52
   3  15  27   6
  14  25   2  10
```

El primer despliegue del arreglo `val` producido por el programa 11.4 se construye al designar en forma explícita cada elemento del arreglo. El segundo despliegue de valores de elementos del arreglo, el cual es idéntico al primero, se produce usando un ciclo `for` anidado. Los ciclos anidados son útiles en especial cuando se trata con arreglos bidimensionales debido a que le permiten al programador designar y llegar con facilidad a cada elemento. En el programa 11.4, la variable `i` controla el ciclo exterior y la variable `j` controla el ciclo interior. Cada pasada por el ciclo exterior corresponde a una sola fila, con el ciclo interior suministrando los elementos de columna apropiados. Después que se imprime una fila completa, se empieza una línea nueva para la siguiente fila. El efecto es un despliegue del arreglo en una forma fila por fila.



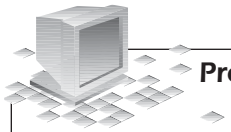
Programa 11.4

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{

  const int NUMFILAS = 3;
  const int NUMCOLS = 4;

  int i, j;
  int val[NUMFILAS][NUMCOLS] = {8, 16, 9, 52, 3, 15, 27, 6, 14, 25, 2, 10};

  cout << "\nDespliegue del arreglo val por elemento explícito"
       << endl << setw(4) << val[0][0] << setw(4) << val[0][1]
       << setw(4) << val[0][2] << setw(4) << val[0][3]
       << endl << setw(4) << val[1][0] << setw(4) << val[1][1]
       << setw(4) << val[1][2] << setw(4) << val[1][3]
       << endl << setw(4) << val[2][0] << setw(4) << val[2][1]
       << setw(4) << val[2][2] << setw(4) << val[2][3];

  cout << "\n\nDespliegue del arreglo val usando un ciclo for anidado";

  for (i = 0; i < NUMFILAS; i++)
  {
    cout << endl;L       // imprime una linea nueva para cada fila
    for (j = 0; j < NUMCOLS; j++)
      cout << setw(4) <<  val[i][j];
  }

  cout << endl;

  return 0;
}
```

Una vez que se han asignado los elementos del arreglo bidimensional, puede comenzar el procesamiento del arreglo. Por lo general, los ciclos `for` se usan para procesar arreglos bidimensionales porque, como se señaló antes, permiten al programador designar y llegar con facilidad a cada elemento del arreglo. Por ejemplo, el ciclo `for` anidado ilustrado en el programa 11.5 se usa para multiplicar cada elemento en el arreglo `val` por el número escalar 10 y desplegar el valor resultante.

A continuación se muestra la salida producida por el programa 11.5:

```
Despliegue de elementos multiplicados
   80  160   90  520
   30  150  270   60
  140  250   20  100
```



Programa 11.5

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  const int NUMFILAS = 3;
  const int NUMCOLS = 4;

  int i, j;
  int val[NUMFILAS][NUMCOLS] = {8, 16, 9, 52,
                                3,15,27,6,
                               14,25,2,10};

// multiplica cada elemento por 10 y lo despliega
  cout << "\nDespliegue de elementos multiplicados";
  for (i = 0; i < NUMFILAS; i++)
  {
    cout << endl; // empieza cada fila en una linea nueva
    for (j = 0; j < NUMCOLS; j++)
    {
      val[i][j] = val[i][j] * 10;
      cout << setw(5) << val[i][j];
    }  // fin del ciclo interior
  }    // fin del ciclo exterior
  cout << endl;

  return 0;
}
```

Arreglos dimensionales mayores

Aunque los arreglos con más de dos dimensiones no se usan por lo común, C++ permite que se declare cualquier número de dimensiones. Esto se hace enlistando el tamaño máximo de todas las dimensiones para el arreglo. Por ejemplo, la declaración `int respuesta[4][10][6];` declara un arreglo tridimensional. El primer elemento en el arreglo es designado como respuesta `[0][0][0]` y el último elemento como `respuesta[3][9][5]`.

Desde el punto de vista conceptual, como se ilustra en la figura 11.9, un arreglo tridimensional puede verse como un libro de tablas de datos. Usando esta visualización, el primer índice puede considerarse como la ubicación de la fila deseada en una tabla, el segundo valor índice como la columna deseada y el tercer valor índice, el cual con frecuencia se llama "rango", como el número de página de la tabla seleccionada.

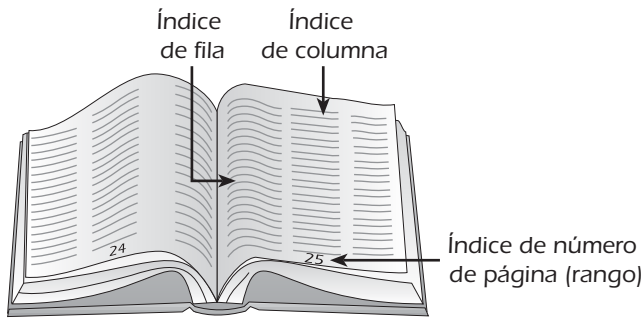


Figura 11.9 *Representación de un arreglo tridimensional.*

Del mismo modo, pueden declararse arreglos de cualquier dimensión. Desde el punto de vista conceptual, un arreglo tetradimensional puede representarse como un anaquel de libros, donde la cuarta dimensión se usa para declarar un libro deseado en el anaquel, y un arreglo pentadimensional puede verse como un librero lleno de libros en el que la quinta dimensión se refiere al anaquel seleccionado en el librero. Usando la misma analogía, un arreglo de seis dimensiones puede considerarse como una hilera de libreros donde la sexta dimensión hace referencia al librero deseado en la hilera; un arreglo de siete dimensiones puede considerarse como múltiples hileras de libreros donde la séptima dimensión hace referencia a la hilera deseada, etc. De manera alternativa, los arreglos de tres, cuatro, cinco, seis, etc. dimensiones pueden verse como *n* múltiplos matemáticos del orden de tres, cuatro, cinco, seis, etc., respectivamente.

Ejercicios 11.3

1. Escriba instrucciones de especificación apropiadas para
   - **a.** un arreglo de números enteros con 6 filas y 10 columnas
   - **b.** un arreglo de números enteros con 2 filas y 5 columnas
   - **c.** un arreglo de caracteres con 7 filas y 12 columnas
   - **d.** un arreglo de caracteres con 15 filas y 7 columnas
   - **e.** un arreglo de números de precisión doble con 10 filas y 25 columnas
   - **f.** un arreglo de números de precisión doble con 16 filas y 8 columnas

2. Determine la salida producida por el siguiente programa:

```
#include <iostream>
using namespace std;

int main()
{
  int i, j, val[3][4] = {8,16,9,52,3,15,27,6,14,25,2,10};
```

```
  for (i = 0; i < 3; ++i)
    for (j = 0; j < 4; ++j)
      cout << "  " << val[i][j];

  return 0;
}
```

3. a. Escriba un programa en C++ que sume los valores de todos los elementos en el arreglo val usados en el ejercicio 2 y despliegue el total.
b. Modifique el programa escrito para el ejercicio 3a para desplegar el total de cada fila por separado.

4. Escriba un programa en C++ que sume elementos equivalentes de los arreglos bidimensionales llamados `primero` y `segundo`. Ambos arreglos deberán tener dos filas y tres columnas. Por ejemplo, el elemento `[1][2]` del arreglo resultante deberá ser la suma de `primero[1][2]` y `segundo[1][2]`. Los arreglos `primero` y `segundo` deberán inicializarse como sigue:

```
     PRIMERO              SEGUNDO
16    18    23        24    52    77
54    91    11        16    19    59
```

5. a. Escriba un programa en C++ que encuentre y despliegue el valor máximo en un arreglo bidimensional de números enteros. El arreglo deberá ser declarado como un arreglo de números enteros de 4 por 5 e inicializarse con los datos 16, 22, 99, 4, 18, −258, 4, 101, 5, 98, 105, 6, 15, 2, 45, 33, 88, 72, 16, 3
b. Modifique el programa escrito en el ejercicio 5a de modo que también despliegue los números de los subíndices de fila y columna del valor máximo.

6. Escriba un programa en C++ para seleccionar los valores en un arreglo de cuatro por cinco de números enteros positivos en orden creciente y almacenar los valores seleccionados en el arreglo unidimensional llamado **ordenar**. Use la instrucción de datos dada en el ejercicio 5a para inicializar el arreglo bidimensional.

7. a. Un ingeniero ha construido un arreglo bidimensional de números reales que tiene 3 filas y 5 columnas. Este arreglo contiene en la actualidad los voltajes de prueba de un amplificador. Escriba un programa en C++ que introduzca de manera interactiva 15 valores del arreglo y luego determine el número total de voltajes en los rangos menor que 60, mayor que o igual a 60 y menor que 70, mayor que o igual a 70 y menor que 80, mayor que o igual a 80 y menor que 90, y mayor que o igual a 90.
b. Introducir 15 voltajes cada vez que se ejecuta el programa escrito para el ejercicio 7a es engorroso. Por consiguiente, ¿qué método es apropiado para inicializar el arreglo durante la fase de prueba?
c. ¿Cómo podría modificarse el programa que escribió para el ejercicio 7a para incluir el caso de que no haya ningún voltaje presente? Es decir, ¿qué voltaje podría usarse para indicar un voltaje inválido y cómo tendría que modificarse su programa para excluir el conteo de dicho voltaje?

11.4 APLICACIONES

Los arreglos son muy útiles en cualquier aplicación que requiera múltiples recorridos por el mismo conjunto de elementos de datos. Dos de dichas aplicaciones se presentan en esta sección. La primera aplicación es un análisis estadístico de datos que requiere dos recorridos por los datos. El primer recorrido se usa para introducir y determinar el promedio de los datos, mientras el segundo recorrido usa el promedio para determinar una desviación estándar. La segunda aplicación presenta un método simple pero elegante de graficar datos en una pantalla de video o en una impresora estándar. Aquí se usa un primer recorrido para inicializar el arreglo con los puntos de datos que se van a graficar y para determinar los valores mínimos y máximo del arreglo. Estos valores se usan luego para calcular un factor de escala apropiado para asegurar que la gráfica final cabe dentro del área de la pantalla de video o el papel. Por último, se hace un segundo recorrido por el arreglo para producir la gráfica.

Aplicación 1: Análisis estadístico

Se va a desarrollar un programa que acepte una lista de un máximo de 100 voltajes como entrada, determine tanto el promedio como la desviación estándar de los voltajes introducidos, y luego despliegue los resultados.

Paso 1 Analizar el problema

El planteamiento del problema indica que se requieren dos valores de salida: un promedio y una desviación estándar. En el paso 2 se verificará que se sabe cómo se calculan estos valores. El elemento de entrada definido en el planteamiento del problema es un máximo de 100 voltajes. Esto significa que cualquier número de voltajes, de cero a 100 podría ser introducido por el usuario cuando se ejecute el programa. Para facilitar esto, se tendrá que preguntar al usuario cuántos voltajes pretende introducir. Por tanto, la primera entrada será el número de voltajes que se introducirán.

Paso 2 Desarrollar una solución

Las especificaciones de E/S determinadas en el paso 1 definen que el usuario introducirá dos tipos de entradas: el número de voltajes seguido por los datos reales. Con base en esta entrada el programa calculará y desplegará el promedio y la desviación estándar de los datos. Estos elementos de salida se determinan como sigue:

Calcular el promedio sumando los voltajes y dividiéndolos entre el número de voltajes que se sumaron
Determinar la desviación estándar

1. ***Restando el promedio de cada voltaje individual: esto produce un conjunto de miembros nuevos, cada uno de los cuales se llama desviación***
2. ***Elevar al cuadrado cada desviación encontrada en el paso anterior***
3. ***Sumar las desviaciones cuadradas y dividir la suma entre el número de desviaciones***
4. ***La raíz cuadrada del número encontrado en el paso anterior es la desviación estándar***

Hay que observar que el cálculo de la desviación estándar requiere el promedio, lo cual significa que la desviación estándar sólo puede ser calculada después que se ha calculado el promedio. Ésta es la ventaja de especificar el algoritmo, en detalle, antes de realizar cualquier codificación; asegura que todas las entradas y todos los requisitos necesarios se descubren pronto en el proceso de programación.

Para asegurar que se entiende el procesamiento requerido, se hace un cálculo manual. Para este cálculo se supondrá de manera arbitraria que se van a determinar el promedio y la desviación estándar de los siguientes 10 voltajes: 98, 82, 67, 54, 78, 83, 95, 76, 68 y 63.

El promedio de estos datos se determina como

$$\begin{aligned}\text{Promedio} &= (98 + 82 + 67 + 54 + 78 + 83 + 95 + 76 + 68 + 63)/10\\ &= 76.4\end{aligned}$$

La desviación estándar se calcula determinando primero la suma de las desviaciones cuadradas. Entonces se obtiene la desviación estándar dividiendo la suma resultante entre 10 y extrayendo su raíz cuadrada.

$$\begin{aligned}\text{Suma de desviaciones cuadradas} &= (98 - 76.4)^2\\ &\quad + (82 - 76.4)^2\\ &\quad + (67 - 76.4)^2\\ &\quad + (54 - 76.4)^2\\ &\quad + (78 - 76.4)^2\\ &\quad + (83 - 76.4)^2\\ &\quad + (95 - 76.4)^2\\ &\quad + (76 - 76.4)^2\\ &\quad + (68 - 76.4)^2\\ &\quad + (63 - 76.4)^2\\ &= 1730.4007\end{aligned}$$

$$\begin{aligned}\text{Desviación estándar} &= \sqrt{1730.4007 / 10}\\ &= \sqrt{173.04007}\\ &= 13\,154470\end{aligned}$$

Habiendo especificado el algoritmo requerido de cada función, ahora se está en posición de codificarlos.

Paso 3 Codificar la solución

El programa 11.6 presenta la versión en C++ del algoritmo seleccionado. Hay que observar que el programa usa un ciclo `for` para introducir y sumar los voltajes individuales y un segundo ciclo `for` para determinar la suma de las desviaciones cuadradas. Debido a que el cálculo de las desviaciones cuadradas requiere el promedio, la desviación estándar sólo puede calcularse después que se ha calculado el promedio. Obsérvese también que el valor de terminación del contador del ciclo en ambos ciclos `for` es `numvoltios`, el cual es el número de voltajes introducidos por el usuario. El uso de este argumento le da al programa su generalidad y permite que sea usado en listas de cualquier número de voltajes, hasta `NUMELS`, el cual es definido como 100 por la instrucción `const`.

Programa 11.6

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{

  const int NUMELS = 100;

  int i, numvoltios;
  double voltio[NUMELS];
  double promedio, desvest;
  double sumavoltios = 0.0, sumdesv = 0.0;

  cout << "Introduzca el número de voltajes que se va a analizar: ";
  cin >> numvoltios;

  // lee los voltajes de entrada y los totaliza
  for (i = 0; i < numvoltios; i++)
  {
      cout << "Introduzca voltaje " << i+1 << ": ";
      cin >> voltio[i];
    sumvoltios = sumvoltios + voltio[i];
  }
  // calcula y despliega el promedio
  promedio = sumvoltios / numvoltios;
  cout << "\nEl promedio de los voltajes es "
       << setw(11) << setiosflags(ios::showpoint)
       << setprecision(8) << promedio << endl;

  // calcula y despliega la desviación estándar
  for (i = 0; i < numvoltios; i++)
     sumadesv = sumadesv + pow(voltio[i] - promedio),2);
  desvest = sqrt(sumadesv/numvoltios);
  cout << "La desviación estándar de los voltajes es "
       << setw(11) << setiosflags(ios::showpoint)
       << setprecision(8) << desvest << endl;

  return 0;

}
```

Paso 4 Prueba y corrección del programa

Una ejecución de prueba usando el programa 11.6 produjo el siguiente despliegue:

```
Introduzca el número de voltajes que se va a analizar: 10
Introduzca voltaje 1: 98
Introduzca voltaje 2: 82
Introduzca voltaje 3: 67
Introduzca voltaje 4: 54
Introduzca voltaje 5: 78
Introduzca voltaje 6: 83
Introduzca voltaje 7: 95
Introduzca voltaje 8: 76
Introduzca voltaje 9: 68
Introduzca voltaje 10: 63

El promedio de los voltajes es 76.400000
La desviación estandar de los voltajes es 13.154467
```

Aunque este resultado concuerda con nuestro cálculo manual previo, la prueba en realidad no está completa sin verificar el cálculo en los puntos límite. En este caso dicha prueba consiste en comprobar el cálculo con todos los valores iguales, como todos 0 y todos 100. Otra prueba simple sería usar cinco 0 y cinco 100. Se dejan estas pruebas como ejercicio.

Aplicación 2: Mantenimiento de una lista[3]

Un problema de programación común es mantener una lista en orden numérico o alfabético. Por ejemplo, los números de partes de inventario se mantienen por lo general en orden numérico, pero las listas telefónicas se conservan en orden alfabético.

Para esta aplicación, escriba una función que inserte un código de número de parte de tres dígitos dentro de una lista de números de parte. La lista se mantiene en orden numérico creciente y no se permiten códigos de número de parte duplicados. Se tiene que asignar un tamaño máximo de la lista de 100 valores y se usará un valor centinela de 9999 para indicar el final de la lista. Por ejemplo, si la lista actual contiene nueve códigos de número de parte, la décima posición en la lista contendrá el valor centinela.

Paso 1 Analizar el problema

La salida requerida es una lista actualizada de códigos de tres dígitos en la cual el código nuevo ha sido insertado en la lista existente. Los elementos introducidos para esta función son el arreglo existente de códigos de identificación y el nuevo código que se ha insertado en la lista.

[3]Este tema requiere una comprensión de la transmisión de arreglos a una función (véase la sección 11.5).

Paso 2 Desarrollar una solución

La inserción de un número de parte nuevo en la lista existente requiere el siguiente procedimiento:

Determinar en qué parte de la lista se colocará el código nuevo
Esto se hace comparando el código nuevo con cada valor en la lista actual hasta que se encuentre una coincidencia, se localiza un código de identificación más grande que el código nuevo o se encuentra el final de la lista
Si el código nuevo coincide con un código existente,
despliega un mensaje de que el código existe
Si no
Para hacer espacio para el nuevo elemento en el arreglo,
se mueve cada elemento una posición hacia abajo. Esto se hace comenzando desde el valor centinela y moviendo cada elemento hacia abajo una posición hasta que queda desocupada la posición deseada en la lista
Se inserta el código nuevo en la posición desocupada
Termina el si

Para asegurar que se entiende este algoritmo, se hará un cálculo manual. Para este cálculo, suponga que la lista de códigos de identificación consiste de los números mostrados en la figura 11.10a. Si el código número 142 se va a insertar en esta lista, debe colocarse en la cuarta posición en la lista después del número 136. Para hacer espacio para el código nuevo, todos los códigos de la cuarta posición hasta el final de la lista deben moverse una posición hacia abajo como se ilustra en la figura 11.10b. El movimiento siempre se inicia desde el final de la lista y procede desde el valor centinela hacia atrás hasta alcanzar la posición deseada en la lista. (Si la copia procediera hacia delante desde el cuarto elemento, el número 144 sería reproducido en todas las ubicaciones subsiguientes hasta alcanzar el valor centinela.) Después del movimiento de los elementos necesarios, se inserta el nuevo código en la posición correcta. Esto crea la lista actualizada que se muestra en la figura 11.10c.

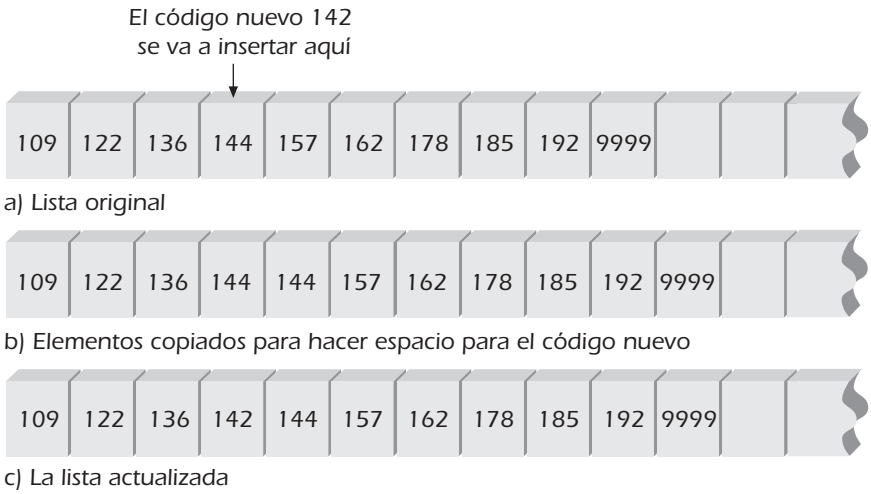


Figura 11.10 Actualización de una lista ordenada de números de identificación.

Paso 3 Codificar la solución

Para este problema se usa el nombre de argumento `idcodigo` para el arreglo transmitido de números de identificación y el nombre de argumento `codigonuevo` para el número de código nuevo que se va a insertar en el arreglo. Aquí, el arreglo transmitido se usa para recibir el arreglo original de números y como arreglo actualizado final. En forma interna respecto a la función, se usará una variable llamada `posnueva` para contener la posición en la lista donde se va a insertar el código nuevo y la variable llamada `posfinal` para contener el valor de la posición del centinela. La variable `i` se usará como un valor índice.

Usando estos nombres de argumento y variable, la función llamada `insert()` ejecuta el procesamiento requerido. Después de aceptar el arreglo y el valor del código nuevo como argumentos, `insert()` ejecuta las cuatro tareas principales descritas en el seudocódigo seleccionado en el paso 2.

```
void insert(int idcodigo[], int codigonuevo)
{
  int i, posnueva, posfinal;

  // encuentra la posición correcta para insertar el código nuevo
  i = 0;
  while (idcodigo[i] < codigonuevo)
    i++;
  if (idcodigo[i] == codigonuevo)
    cout << "\nEste código de identificacion ya esta en la lista";
  else
  {
    posnueva = i;  // encuentra la posición para el código nuevo

    // encuentra el final de la lista
    while (idcodigo[i] != 9999)
      i++;
    posfinal = i;

    // mueve idcodigos una posición
    for (i = posfinal; i >= posnueva; --i)
      idcodigo[i+1] = idcodigo[i];

    // inserta el código nuevo
    idcodigo[posnueva] = codigonuevo;
  }
}
```

La primera tarea de la función es determinar la posición correcta del código nuevo. Esto se hace recorriendo la lista en tanto cada valor encontrado sea menor que el código nuevo. En vista que el valor centinela de 9999 es mayor que cualquier código nuevo, el ciclo debe detenerse cuando se alcance el valor centinela.

Después que se determina la posición correcta, se encuentra la posición del valor centinela, la cual es el último elemento en la lista. Empezando desde esta última posición, cada elemento en la lista se mueve una posición hasta que se alcanza el valor en la posición nueva requerida. Por último, el nuevo código de identificación se inserta en la posición correcta.

Paso 4 Prueba y depuración de la función

El programa 11.7 incorpora la función `insert()` dentro de un programa completo. Esto permite probar la función con los mismos datos usados en el cálculo manual.

Programa 11.7

```
#include <iostream>
using namespace std;

void insert(int [], int);      // prototipo de la función

int main()
{
  const int NUM_MAX = 100;
  int codigonuevo, i;
  int id[NUM_MAX] = {109, 122, 136, 144, 157, 162, 178, 185, 192, 9999};

  cout << "\nIntroduzca el código de identificación nuevo: ";
  cin >> codigonuevo;

  insert(id, codigonuevo);

  cout << "\nLa lista actualizada es:";
  i = 0;
  while(id[i] != 9999)
  {
    cout << " " << id[i];
    i++;
  }
  cout << endl;

  return 0;
}
```

(Continúa)

(Continuación)

```
void insert(int idcodigo[], int codigonuevo)
{
  int i, posnueva, posfinal;

  // encuentra la posición correcta para insertar el código nuevo
  i = 0;
  while (idcodigo[i] < códigonuevo)
    i++;
  if (idcodigo[i] == codigonuevo)
    cout << "\nEste código de identificación ya está en la lista";
  else
  {
    posnueva = i;    // encuentra la posición para el código nuevo

    // encuentra el final de la lista
    while (idcodigo[i] != 9999)
      i++;
    posfinal = i;

    // mueve idcodigos una posición
    for (i = posfinal; i >= posnueva; i--)
      idcodigo[i+1] = idcodigo[i];

    // inserta el código nuevo
    idcodigo[posnueva] = codigonuevo;
  }

  return;
}
```

A continuación se presenta una muestra de la ejecución del programa 11.7:

```
Introduzca el código de identificación nuevo: 142
La lista actualizada es: 109 122 136 142 144 157 162 178 185 192
```

Aunque este resultado concuerda con el cálculo manual previo, no constituye la prueba completa del programa. Para estar seguros que el programa funciona en todos los casos, deberán hacerse ejecuciones de prueba que realicen lo siguiente:

1. Introducir un código de identificación nuevo que duplique un código existente
2. Colocar un código de identificación nuevo al principio de la lista
3. Colocar un código de identificación nuevo al final de la lista

Por último, la restricción del tamaño máximo del arreglo de 100 números de partes designado en la especificación original del problema es poco realista en la práctica. Por lo general, el tamaño máximo nunca se conoce con certeza debido a que cambian condiciones que hacen poco realista el tamaño original. La solución para esta incertidumbre es declarar un arreglo grande que anticipe los posibles cambios, lo cual desperdicia memoria de la computadora, o crear un arreglo que pueda expandirse de manera dinámica y automática conforme se agreguen números de parte nuevos. En la sección 11.6 se presenta la forma en que puede implementarse esta segunda solución.

Ejercicios 11.4

1. Introduzca y ejecute el programa 11.6 en su computadora.
2. Ejecute el programa 11.6 para determinar el promedio y la desviación estándar de la siguiente lista de 15 voltajes: 68, 72, 78, 69, 85, 98, 95, 75, 77, 82, 84, 91, 89, 65, 74.
3. Introduzca y ejecute el programa 11.7 en su sistema de cómputo.
4. **a.** Pruebe el programa 11.7 usando un número de parte de 86, lo cual deberá colocar este código nuevo al principio de la lista existente.

   b. Pruebe el programa 11.7 usando un número de parte de 200, lo cual deberá colocar este número de parte nuevo al final de la lista existente.
5. **a.** Determine un algoritmo para eliminar una entrada de una lista ordenada de números.

   b. Escriba una función, `delete()`, la cual usa el algoritmo seleccionado en el ejercicio 5a, para eliminar un código de identificación de la lista de números ilustrados en la figura 11.10a.
6. Suponga que las siguientes letras son almacenadas en un arreglo de alfabeto: B, J, K, M, S, Z. Escriba y pruebe una función, `agregarletra()`, que acepte el arreglo de alfabeto y una letra nueva como argumentos y luego inserte la letra nueva en el orden alfabético correcto en el arreglo de alfabeto.

11.5 ARREGLOS COMO ARGUMENTOS

Los elementos del arreglo individuales son transmitidos a una función llamada de la misma manera que las variables escalares individuales; tan sólo se incluyen como variables subindexadas cuando se hace la llamada a la función. Por ejemplo, la llamada a la función

```
hallarMin(voltios[2], voltios[6]);
```

transmite los valores de los elementos `voltios[2]` y `voltios[6]` a la función `hallarMin()`.

Transmitir un arreglo completo de valores a una función es en muchos aspectos una operación más fácil que transmitir elementos individuales. La función llamada recibe acceso al arreglo real, en lugar de a una copia de los valores en el arreglo. Por ejemplo, si `voltios` es un arreglo, la llamada a la función `hallarMax(voltios)`; hace que el arreglo `voltios` completo esté disponible para la función `hallarMax()`. Esto es diferente al transmitir una sola variable a una función.

Se recordará que cuando se transmite un solo argumento escalar a una función, la función llamada sólo recibe una copia del valor transmitido, el cual es almacenado en uno de los parámetros de la función. Si los arreglos fueran transmitidos de esta manera, tendría que crearse una copia del arreglo completo. Para arreglos grandes, hacer copias duplicadas del arreglo para cada llamada a la función sería un desperdicio de almacenamiento de computadora y frustraría el esfuerzo de devolver cambios a elementos múltiples hechos por el programa invocado. (Se recordará que una función devuelve cuando mucho un valor directo.) Para evitar estos problemas, a la función llamada se le da acceso directo al arreglo original.[4] Por tanto, los cambios hechos por la función llamada se hacen en forma directa al arreglo en sí. Para los siguientes ejemplos específicos de llamadas a función, suponga que los arreglos `nums, claves, voltios` y `corriente` son declarados como:

```
int nums[5];              // un arreglo de cinco números enteros
char claves[256];         // un arreglo de 256 caracteres
double voltios[500], corriente[500];   // dos arreglos de 500
                                          números de precisión
                                          doble
```

Para estos arreglos, pueden hacerse las siguientes llamadas a función:

```
hallarMax(nums);
hallarCh(claves);
calcTot(nums, voltios, corriente);
```

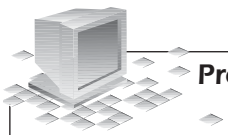
En cada caso, la función llamada recibe acceso directo al arreglo nombrado.

En el lado receptor, la función llamada debe ser alertada de que un arreglo está disponible. Por ejemplo, son líneas adecuadas para el encabezado de la función para las funciones previas

```
int hallarMax(int vals[5])
char hallarCh(char in_claves[256])
void calcTot(int arr1[5], double arr2[500], double arr3[500])
```

En cada una de estas líneas de encabezado de la función, los nombres en la lista de parámetros son elegidos por el programador. Sin embargo, los nombres de parámetros usados por las funciones todavía se refieren al arreglo original creado fuera de la función. Esto queda claro en el programa 11.8.

[4]Esto se logra debido a que la dirección inicial del arreglo en realidad se transmite como un argumento. El parámetro formal que recibe este argumento de dirección es un apuntador. La relación íntima entre los nombres de arreglos y apuntadores se presenta en el capítulo 12.



Programa 11.8

```
#include <iostream>
using namespace std;

const int MAXELS = 5;
int hallarMax(int [MAXELS]);       // prototipo de la función

int main()
{
  int nums[MAXELS] = {2, 18, 1, 27, 16};

  cout << "El valor máximo es " << hallarMax(nums) << endl;

  return 0;
}

// halla el valor máximo
int hallarMax(int vals[MAXELS])
{
  int i, max = vals[0];
  for (i = 1; i < MAXELS; i++)
    if (max < vals[i]) max = vals[i];

  return max;
}
```

Hay que observar que el prototipo de la función para `hallarMax()` declara que `hallarMax` devolverá un número entero y espera un arreglo de cinco números enteros como un argumento real. También es importante saber que sólo se crea un arreglo en el programa 11.9. En `main()` este arreglo se conoce como `nums`, y en `hallarMax` el arreglo se conoce como `vals`. Como se ilustra en la figura 11.11, ambos nombres se refieren al mismo arreglo. Por tanto, en la figura 11.11 `vals[3]` es el mismo elemento que `nums[3]`.

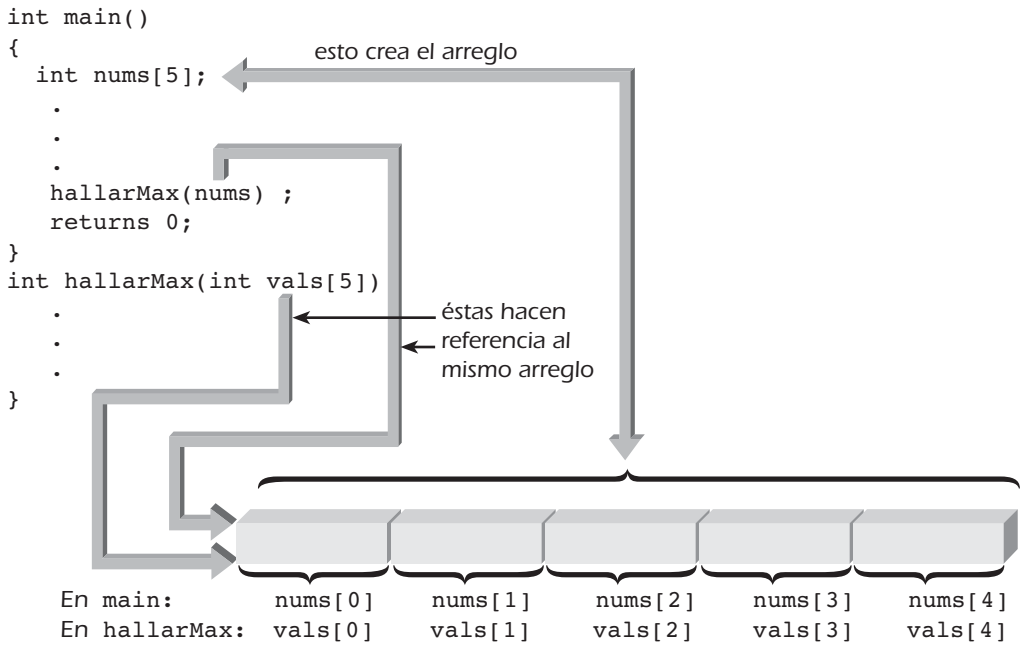


Figura 11.11 *Sólo se crea un arreglo.*

La declaración de parámetros en la línea de encabezado `hallarMax()` en realidad contiene información extra que la función no requiere. Todo lo que debe saber `hallarMax()` es que el parámetro `vals` hace referencia a un arreglo de números enteros. Dado que el arreglo se ha creado en `main()` y no se necesita espacio de almacenamiento adicional en `hallarMax()`, la declaración para `vals` puede omitir el tamaño del arreglo. Por tanto, una línea de encabezado de función alternativa es

```
int hallarMax(int vals[])
```

Esta forma del encabezado de la función tiene más sentido cuando nos damos cuenta que sólo un elemento es transmitido en realidad a `hallarMax` cuando se llama a la función, la dirección inicial del arreglo `nums`. Esto se ilustra en la figura 11.12.

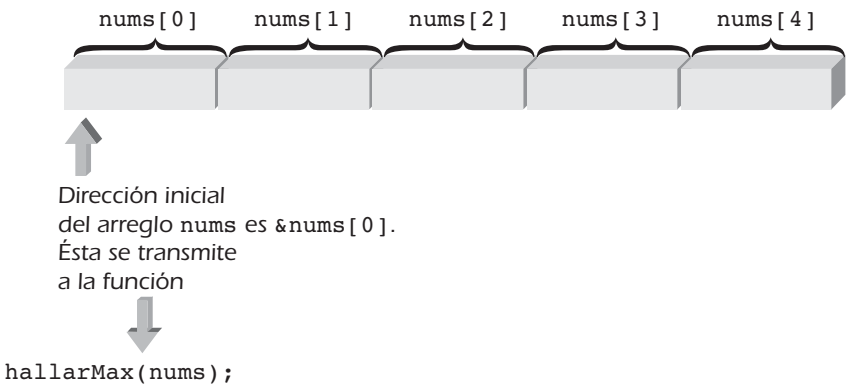


Figura 11.12 *Se transmite la dirección inicial del arreglo.*

En vista que sólo la dirección inicial de `vals` es transmitida a `hallarMax`, el número de elementos en el arreglo no necesita incluirse en la declaración para `vals`.[5] De hecho, por lo general es aconsejable omitir el tamaño del arreglo en la línea de encabezado de la función. Por ejemplo, considérese la forma más general de `hallarMax()`, la cual puede usarse para encontrar el valor máximo de un arreglo de números enteros de tamaño arbitrario.

```
int hallarMax(int vals[], int numels)   // encuentra el valor máximo
{
  int i, max = vals[0];

  for (i = 1; i < numels; i++)
   if (max < vals[i])
     max = vals[i];

  return max;
}
```

La forma más general de `hallarMax()` declara que la función devuelve un valor entero. La función espera la dirección inicial de un arreglo de números enteros y el número de elementos en el arreglo como argumentos. Entonces, usando el número de elementos como el límite para su búsqueda, el ciclo `for` de la función causa que cada elemento del arreglo sea examinado en orden secuencial para localizar el valor máximo. El programa 11.9 ilustra el uso de `hallarMax()` en un programa completo.

La salida desplegada tanto por el programas 11.8 como por el 11.9 es

```
El valor máximo es 27
```

Transmitir arreglos bidimensionales a una función es un proceso idéntico a transmitir arreglos unidimensionales. La función llamada recibe acceso al arreglo entero. Por ejemplo, suponiendo que `val` es un arreglo bidimensional, la llamada a la función `desplegar(val);` hace que el arreglo `val` completo esté disponible para la función llamada `desplegar()`. Por tanto, los cambios hechos por `desplegar()` se harán de manera directa al arreglo `val`. Como ejemplos adicionales, suponga que los siguientes arreglos bidimensionales llamados prueba, factores y empuje se declaran como:

```
int prueba[7][9];
float factores[26][10];
double empuje[256][52];
```

entonces las siguientes llamadas a función son válidas:

```
hallarMax(prueba);
obtener(factores);
promedio(empuje);
```

[5]Una consecuencia importante de esto es que `hallarMax()` tiene acceso directo al arreglo transmitido. Esto significa que cualquier cambio a un elemento del arreglo `vals` es un cambio al arreglo `nums`. Esto es diferente de manera significativa de la situación con las variables escalares, donde la función llamada no recibe acceso directo a la variable transmitida.

Programa 11.9

```
// este programa despliega un mensaje
#include <iostream>
using namespace std;

int hallarMax(int [], int);    // prototipo de la función

int main()
{
  const int MAXELS = 5;
  int nums[MAXELS] = {2, 18, 1, 27, 16};

  cout << "El valor máximo es "
       << hallarMax(nums, MAXELS) << endl;

  return 0;
}

// encuentra el valor máximo
int hallarMax(int vals[], int numels)
{
  int i, max = vals[0];

  for (i = 1; i < numels; i++)
    if (max < vals[i]) max = vals[i];

  return max;
}
```

En el lado receptor, debe alertarse a la función llamada que un arreglo bidimensional queda disponible. Por ejemplo, suponiendo que cada una de las funciones anteriores devuelve un número entero, son líneas de encabezado de función adecuadas para estas funciones:

```
int hallarMax(int nums[7][9])
int obtener(float valores[26][10])
int promedio(double vals[256][52])
```

En cada una de estas líneas de encabezado de función, los nombres de parámetros elegidos se usarán de manera interna en el cuerpo de la función. Sin embargo, estos nombres de parámetros se referirán aún al arreglo original creado fuera de la función. El programa 11.10 ilustra la transmisión de un arreglo bidimensional a una función que despliega los valores del arreglo.

Programa 11.10

```
#include <iostream>
#include <iomanip>
using namespace std;

const int FILAS = 3;
const int COLS = 4;
void desplegar(int [FILAS][COLS]);    // prototipo de la función

int main()
{
  int val[FILAS][COLS] = {8,16,9,52,
                          3,15,27,6,
                         14,25,2,10};

  desplegar(val);

  return 0;
}

void desplegar(int nums[FILAS][COLS])
{
  int num_fila, num_col;
  for (num_fila = 0; num_fila < FILAS; num_fila++)
  {
    for (num_col = 0; num_col < COLS; num_col++)
      cout << setw(4) << nums[num_fila][num_col];
    cout << endl;
  }

  return;
}
```

Sólo un arreglo se crea en el programa 11.10. Este arreglo se conoce como `val` en `main()` y como `nums` en `desplegar()`. Por tanto, `val[0][2]` se refiere al mismo elemento que `nums[0][2]`.

Hay que observar el uso del ciclo `for` anidado en el programa 11.10 para recorrer cada elemento del arreglo. En el programa 11.10, la variable `num_fila` controla el ciclo exterior y la variable `num_col` controla el ciclo interior. Para cada recorrido del ciclo exterior, el cual corresponde a una sola fila, el ciclo interior hace un recorrido por los elementos de columna. Después que se imprime una fila completa, se inicia una línea nueva para la siguiente fila. El efecto es un despliegue del arreglo en una forma fila por fila:

```
   8  16   9  52
   3  15  27   6
  14  25   2  10
```

La declaración de parámetros para `nums` en `desplegar()` contiene información adicional que no es requerida por la función. La declaración para `nums` puede omitir el tamaño de la filas del arreglo. Por tanto, un prototipo de la función alternativo es

```
desplegar(int nums[][4]);
```

La razón por la que debe incluirse el tamaño de las columnas mientras el tamaño de las filas es opcional se hace evidente cuando se considera cómo se almacenan los elementos del arreglo en la memoria. Empezando con el elemento `val[0][0]`, cada elemento subsiguiente se almacena de manera consecutiva, fila por fila, como `val[0][0], val[0][1], val[0][2], val[0][3], val[1][0], val[1][1]`, etc., como se ilustra en la figura 11.13.

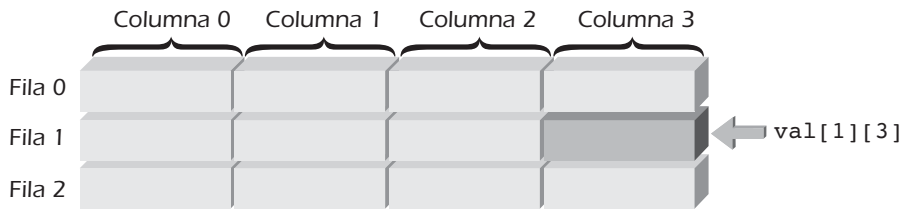
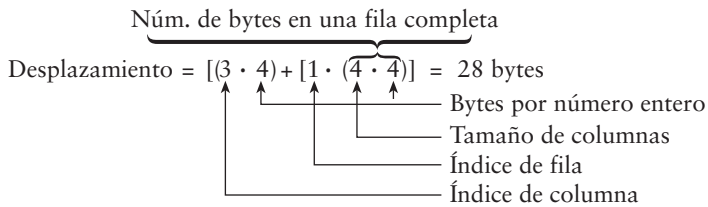


Figura 11.13 *Almacenamiento del arreglo* `val`.

Como ocurre con todos los accesos a arreglos, un elemento individual del arreglo `val` se obtiene sumando un desplazamiento a la ubicación inicial del arreglo. Por ejemplo, el elemento `val[1][3]` del arreglo `val` ilustrado en la figura 11.13 se localiza con un desplazamiento de 14 bytes desde el inicio del arreglo. Internamente, el compilador usa el índice de fila, el índice de columna y el tamaño de las columnas para determinar este desplazamiento usando el siguiente cálculo (una vez más, suponiendo cuatro bytes para un `int`):



El tamaño de las columnas es necesario en el cálculo del desplazamiento, de modo que el compilador pueda determinar el número de posiciones que se va a saltar con el fin de obtener la fila deseada.

Ejercicios 11.5

1. La siguiente declaración se usó para crear el arreglo `voltios`:

   ```
   int voltios[500];
   ```

   Escriba dos líneas de encabezado de función diferentes para una función llamada `ordenarArreglo()` que acepte el arreglo `voltios` como un parámetro llamado `enArreglo`.

2. La siguiente declaración se usó para crear el arreglo `factores`:

```
double factores[256];
```

Escriba dos líneas de encabezado de función diferentes para una función llamada `hallarClave()` que acepte el arreglo factores como un parámetro llamado `seleccionar`.

3. La siguiente declaración se usó para crear el arreglo `potencia`:

```
double potencia[256];
```

Escriba dos líneas de encabezado de función diferentes para una función llamada `principal()` que acepte el arreglo `potencia` como un argumento llamado `tasas`.

4. ***a.*** Modifique la función `hallarMax()` en el programa 11.8 para localizar el valor mínimo del arreglo transmitido.
b. Incluya la función escrita en el ejercicio 4a en un programa completo y ejecute el programa en una computadora.

5. Escriba un programa que tenga una declaración en `main()` para almacenar los siguientes números en un arreglo llamado `temps`: 6.5, 7.2, 7.5, 8.3, 8.6, 9.4, 9.6, 9.8, 10.0. Deberá haber una llamada a la función `mostrar()` que acepte el arreglo `temps` como un parámetro llamado `temps` y luego despliegue los números en el arreglo.

6. Escriba un programa que declare tres arreglos unidimensionales llamados `voltios`, `corriente` y `resistencia`. Cada arreglo deberá declararse en `main()` y deberá ser capaz de contener diez números de precisión doble. Los números que deberán almacenarse en `corriente` son 10.62, 14.89, 13.21, 16.55, 18.62, 9.47, 6.58, 18.32, 12.15, 3.98. Los números que deberán almacenarse en `resistencia` son 4, 8.5, 6, 7.35, 9, 15.3, 3, 5.4, 2.9, 4.8. Su programa deberá transmitir estos tres arreglos a una función llamada `calc_voltios()`, la cual deberá calcular los elementos en el arreglo `voltios` como el producto de los elementos correspondientes en los arreglos `corriente` y `resistencia` (por ejemplo, `voltios[1] = corriente[1] * resistencia[1]`). Después que `calc_voltios()` ha puesto valores en el arreglo `voltios`, los valores en el arreglo deberán desplegarse desde adentro de `main()`.

7. Escriba un programa que incluya dos funciones llamadas `calc_prom()` y `varianza()`. La función `calc_prom()` deberá calcular y devolver el promedio de los valores almacenados en el arreglo llamado `valores_prueba`. El arreglo deberá ser declarado en `main()` e incluir los valores 89, 95, 72, 83, 99, 54, 86, 75, 92, 73, 79, 75, 82, 73. La función `varianza()` deberá calcular y devolver la varianza de los datos. La varianza se obtiene restando el promedio de cada valor en `valores_prueba`, elevando al cuadrado los valores obtenidos, sumándolos y dividiéndolos entre el número de elementos en `valores_prueba`. Los valores devueltos de `calc_promedio()` y `varianza()` deberán desplegarse usando instrucciones `cout` en `main()`.

11.6 LA CLASE DE VECTOR STL

Muchas aplicaciones de programación requieren que se expandan y contraigan listas conforme se agregan y eliminan elementos de la lista. Aunque puede lograrse expandir y contraer un arreglo creando, copiando y eliminando arreglos, esta solución es costosa en función de la programación inicial, mantenimiento y tiempo de prueba. Para satisfacer la necesidad de proporcionar un conjunto de estructuras de datos probado y genérico que pueda modificarse, expandirse y contraerse, C++ proporciona un conjunto útil de clases en su Biblioteca Estándar de Plantillas (STL, por sus siglas en inglés)

Cada clase STL es codificada como una plantilla (véase la sección 6.1) que permite la construcción de una estructura de datos genérica, la cual se conoce como **contenedor**. Los términos **lista** y **colección** son sinónimos de contenedor, con cada término refiriéndose a un conjunto de elementos de datos que forma una unidad o grupo natural. Usando esta definición, un arreglo es un contenedor; sin embargo, un contenedor se proporciona como un tipo integrado en contraste con los contenedores creados usando STL.

En esta sección se presenta la clase contenedora de vector STL. Un vector se parece a un arreglo en que almacena elementos a los que se puede tener acceso usando un índice de números enteros que comience en cero, pero es diferente en que un vector se expandirá de manera automática conforme sea necesario y es proporcionado por diversos métodos (funciones) de clase útiles para operar en el vector. La tabla 11.1 enlista estos métodos de clase de vector, resaltando los métodos que se usarán en el programa de demostración.

Tabla 11.1 Resumen de métodos (funciones) y operaciones de clase de vector

Funciones (métodos de clase) y operaciones	Descripción
`vector<TipoDatos> nombre`	Crea un vector vacío con tamaño inicial dependiente del compilador
`vector<TipoDatos> nombre(fuente)`	Crea una copia del vector fuente
`vector<TipoDatos> nombre(`*n*`)`	Crea un vector de tamaño *n*
`vector<TipoDatos> nombre(`*n*`, `*elem*`)`	Crea un vector de tamaño *n* con cada elemento inicializado como `elem`
`vector<TipoDatos> nombre(src.beg, src.end)`	Crea un vector inicializado con elementos de un contenedor fuente que comienza en `src.beg` y termina en `src.end`
`~vector<TipoDatos>()`	Destruye el vector y todos los elementos que contiene
`nombre[índice]`	Devuelve el elemento en el índice designado, sin comprobación de límites
`nombre.at(índice)`	Devuelve el elemento en el argumento del índice especificado, sin comprobación de límites en el valor del índice
`nombre.front()`	Devuelve el primer elemento en el vector
`nombre.back()`	Devuelve el último elemento en el vector
`dest = src`	Asigna todos los elementos del vector `src` al vector `dest`

Tabla 11.1 Resumen de métodos (funciones) y operaciones de clase de vector (continuación)

Funciones (métodos de clase) y operaciones	Descripción
`nombre.assign(n, elem)`	Asigna *n* copias de `elem`
`nombre.assign(src.begin, src.end)`	Asigna los elementos del contenedor src (no necesita ser un vector), entre el rango `src.begin` y `src.end`, al vector nombre
`insertar(pos, elem)`	Inserta elem en la posición pos
`nombre.insertar(pos, n, elem)`	Inserta *n* copias de elem empezando en la posición *pos*
`nombre.insertar(pos, src.begin, src.end)`	Inserta
`nombre.push_back(elem)`	Añade `elem` al final del vector
`nombre.erase(pos)`	Elimina el elemento en la posición especificada
`nombre.erase(begin, end)`	Elimina los elementos dentro del rango especificado
`nombre.resize(valor)`	Redimensiona el vector a un tamaño mayor, con elementos nuevos instanciados usando el constructor por omisión
`nombre.resize(valor, elem)`	Redimensiona el vector a un tamaño mayor, con elementos nuevos instanciados como `elem`
`nombre.clear()`	Elimina todos los elementos del vector
`nombre.swap(nombreB)`	Intercambia los elementos de los vectores `nombreA` y `nombreB`; puede ejecutarse usando el algoritmo `swap()`
`nombreA == nombreB`	Devuelve un valor `verdadero` booleano si los elementos de `nombreA` son iguales a los elementos de `nombreB`; de lo contrario, devuelve un valor `falso`
`nombreA != nombreB`	Devuelve un valor `falso` booleano si los elementos de `nombreA` son iguales a los elementos de `nombreB`; de lo contrario, devuelve valor `verdadero`; igual que `!(nombreA == nombreB)`
`nombreA < nombreB`	Devuelve un valor `verdadero` booleano si `nombreA` es menor que nombreB; de lo contrario, devuelve un valor `falso`
`nombreA > nombreB`	Devuelve un valor `verdadero` booleano si `nombreA` es mayor que `nombreB`; de lo contrario, devuelve un valor `falso`; igual que `nombreB < nombreA`
`nombreA <= nombreB`	Devuelve un valor `verdadero` booleano si `nombreA` es menor que o igual a `nombreB`
`nombreA >= nombreB`	Devuelve un valor `verdadero` booleano si `nombreA` es mayor que o igual a `nombreB`
`nombre.size()`	Devuelve el tamaño de un vector
`nombre.empty()`	Devuelve un valor verdadero booleano si el vector está vacío; de lo contrario, devuelve un valor falso
`nombre.max_size()`	Devuelve el máximo posible de elementos como un número entero
`nombre.capacity()`	Devuelve el máximo posible de elementos, como un número entero, sin reubicación del vector

Además de los métodos de clase de vector específicos mostrados en la tabla 11.1, los vectores tienen acceso al conjunto completo de funciones STL genéricas, conocidas en STL como algoritmos. La tabla 11.2 resume los más usados de estos algoritmos.

Tabla 11.2 Algoritmos de la Biblioteca Estándar de Plantillas (STL) más usados

Nombre del algoritmo	Descripción
`accumulate`	Devuelve la suma de los números en un rango especificado
`binary_search`	Devuelve un valor booleano de `verdadero` si el valor especificado existe dentro del rango especificado; de lo contrario, devuelve `falso`. Sólo puede usarse en un conjunto de valores ordenado
`copy`	Copia elementos de un rango de origen a un rango de destino
`copy_backward`	Copia elementos de un rango de origen a un rango de destino en una dirección inversa
`count`	Devuelve el número de elementos en un rango especificado que corresponde a un valor especificado
`equal`	Compara los elementos en un rango de elementos, elemento por elemento, con los elementos en un segundo rango
`fill`	Asigna todos los elementos en un rango especificado a un valor especificado
`find`	Devuelve la posición de la primera ocurrencia de un elemento en un rango especificado que tenga un valor especificado si el valor existe. Ejecuta una búsqueda lineal, empezando con el primer elemento en un rango especificado y procede un elemento a la vez hasta que se ha buscado en el rango completo o se ha encontrado el elemento especificado
`max_element`	Devuelve el valor máximo de los elementos en el rango especificado
`min_element`	Devuelve el valor mínimo de los elementos en el rango especificado
`random_shuffle`	Revuelve al azar los valores de los elementos en un rango especificado
`remove`	Elimina un valor especificado dentro de un rango especificado sin cambiar el orden de los elementos restantes
`replace`	Reemplaza cada elemento en un rango especificado que tiene un valor especificado con un valor recién especificado
`reverse`	Invierte los elementos en un rango especificado
`search`	Encuentra la primera ocurrencia de un valor o secuencia de valores especificados dentro de un rango especificado
`sort`	Ordena los elementos en un rango especificado en orden ascendente
`swap`	Intercambia valores de elementos entre dos objetos
`unique`	Elimina elementos adyacentes duplicados dentro de un rango especificado

Hay que observar que hay un algoritmo `swap` (tabla 11.2) y un método de clase de vector `swap` (tabla 11.1). Debido a que el método de clase está dirigido a trabajar de manera específica con su tipo contenedor y por lo general se ejecutará más rápido siempre que una clase contenedora proporcione un método con el mismo nombre como un algoritmo, deberá usarse el método de clase.

Por último, diversos elementos adicionales, llamados iteradores, también son proporcionados por la STL. Los iteradores proporcionan el medio para especificar cuáles elementos en un contenedor han de ser operados o cuando se invoca a un algoritmo. Dos de los iteradores más útiles son devueltos por las funciones de iterador de la STL llamadas `begin()` y `end()`. Éstas son funciones de propósito general que devuelven las posiciones del primer y último elementos en un contenedor, respectivamente.

Punto de información

Cuándo usar un arreglo o un vector

Un arreglo es la estructura de datos de primera elección siempre que se tenga una lista de tipos de datos u objetos primitivos que no tengan que expandirse o contraerse.

Un vector es la estructura de datos de primera elección siempre que se tenga una lista de tipos de datos u objetos primitivos que puedan agruparse como un arreglo, pero que deban expandirse o contraerse.

Siempre que sea posible, siempre deben usarse algoritmos STL para operar en arreglos (véase la sección 12.4) y vectores (descritos en esta sección). Tanto las clases como los algoritmos de la STL proporcionan código verificado y confiable que puede acortar el tiempo de desarrollo del programa.

Para hacer esto más tangible y proporcionar una introducción significativa al uso de una clase contenedora STL, se usará la clase contenedora de vector para crear un vector que contenga una lista de números de partes. Como se verá, un vector es similar a un arreglo en C++, excepto que puede expandirse de manera automática según sea necesario.

El programa 11.11 construye un vector y lo inicializa con números enteros almacenados en un arreglo de números enteros. Una vez que se ha inicializado, se usan varios métodos de vector y algoritmos STL para operar en el vector. De manera específica, se usa un método para cambiar un valor existente, un segundo método se usa para insertar un valor dentro del vector y un tercer método se usa para añadir un valor al final de la lista. Después que se aplica cada método y algoritmo, se emplea un objeto `cout` para desplegar los resultados.

Programa 11.11

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  const int NUMELS = 4;
  int n[] ={136, 122, 109, 146};
  int i;

  // instancia un vector de cadenas usando el arreglo n[]
  vector<int> nums_partes(n, n + NUMELS);
```

(Continúa)

(Continuación)

```
    cout << "\nEl vector inicialmente tiene un tamanio de "
         << int(nums_partes.size()) << ",\n y contiene los elementos:\n";
    for (i = 0; i < int(nums_partes.size()); i++)
      cout << nums_partes[i] << "  ";

    // modifica el elemento en la posicion 4 (es decir, indice = 3) en el vector
    nums_partes[3] = 144;
    cout << "\n\nDespués de reemplazar el cuarto elemento, el vector tiene
            un tamanio de "
         << int(nums_partes.size()) << ",\n y contiene los elementos:\n";
    for (i = 0; i < int(nums_partes.size()); i++)
      cout << nums_partes[i] << "  ";

    // inserta un elemento en el vector en la posición 2 (es decir, indice = 1)
    nums_partes.insert(nums_partes.begin()+1, 142);
    cout << "\n\nDespués de insertar un elemento en la segunda posición,"
         << "\n el vector tiene un tamanio de " << int(nums_partes.size()) << ","
         << " y contiene los elementos:\n";
    for (i = 0; i < int(nums_partes.size()); i++)
      cout << nums_partes[i] << "  ";

    // agrega un elemento al final del vector
    nums_partes.push_back(157);
    cout << "\n\nDespués de agregar un elemento al final de la lista,"
         << "\n el vector tiene un tamanio de " << int(nums_partes.size()) << ","
         << " y contiene los elementos:\n";
    for (i = 0; i < int(nums_partes.size()); i++)
      cout << nums_partes[i] << "  ";

    // ordena el vector
    sort(nums_partes.begin(), nums_partes.end());
    cout << "\n\nDespués de ordenarlos, los elementos del vector son:\n";
    for (i = 0; i < int(nums_partes.size()); i++)
      cout << nums_partes[i] << "  ";

    cout << endl;

    return 0;
}
```

Al revisar el programa 11.11, se puede que se han incluido las cuatro líneas de encabezado `<iostream>, <string>, <vector>` y `<algorithm>` y la instrucción `using namespace std;`. El encabezado `<iostream>` es necesario para crear y usar el flujo `cout`, el encabezado `<string>` se requiere para construir cadenas, el encabezado `<vector>` se usa para crear o mover objetos `vector` y el encabezado `<algorithm>` se requiere para el algoritmo `sort` que se aplica después que se ha completado la adición y reemplazo de elementos de `vector`.

La instrucción en el programa 11.11 usada para crear e inicializar el vector llamado `nums_partes` es la siguiente:

```
vector<int> nums_partes(n, n + NUMELS);
```

Aquí, el vector `nums_partes` es declarado como un `vector` del tipo `int` y es inicializado con elementos del arreglo n, comenzando con el primer elemento del arreglo `(elemento n[0])` y terminando con el último elemento del arreglo, el cual se localiza en la posición `n + NUMELS`. Por tanto, el vector tiene un tamaño suficiente para cuatro valores enteros y se ha inicializado con los números enteros `136, 122, 109` y `146`. El siguiente conjunto de instrucciones en el programa 11.11 despliega los valores iniciales en el vector usando notación de vector subindexada estándar que es idéntica a la notación usada para tener acceso a los elementos del arreglo. Sin embargo, desplegar los valores del vector de esta manera requiere saber cuántos elementos contiene cada vector. Conforme se insertan y eliminan elementos, nos gustaría que el vector rastreara la ubicación del primer y último elementos. Esta capacidad es proporcionada de manera automática por dos métodos suministrados para cada contenedor STL: `begin()` y `end()`.

El siguiente conjunto importante de instrucciones consiste en lo siguiente:

```
// modifica el elemento en la posición 4 (es decir, indice = 3) en
   el vector
  nums_partes[3] = 144;


  // inserta un elemento en el vector en la posición 2 (es decir,
     indice = 1)
  nums_partes.insert(nums_partes.begin()+1, 142);
```

Estas instrucciones se usan para modificar un valor de vector existente e insertar un valor nuevo en el vector. De manera específica, la notación `nums_partes[3]` usa indexación estándar, y el método `insert()` usa un argumento que se conoce en STL como iterador. Dichos argumentos se construyen como desplazamientos usando las funciones `begin()` y `end()`. Además, se tiene que especificar el valor que se va a insertar en la posición designada. Por tanto, `nombres[3]` especifica que se cambiará el cuarto elemento en el vector. (Los vectores, como los arreglos, comienzan en la posición índice 0.) El método `insert()` se usa para insertar el valor entero 142 en la segunda posición del vector. Debido a que el método `begin()` devuelve un valor correspondiente al inicio del vector, agregarle 1 designa la segunda posición en el vector.[6] Es en esta posición donde se inserta el valor nuevo, todos los valores subsiguientes se mueven una posición en el vector y el vector se expande de manera automática para aceptar el valor insertado. En este punto en el programa, el vector `nums_partes` contiene ahora los siguientes elementos:

```
136  142  122  109  144
```

Esta ordenación de valores se obtuvo al reemplazar el valor original 146 con 144 e insertando el 142 en la segunda posición, lo cual en forma automática mueve todos los elementos subsiguientes una posición e incrementa el tamaño total del vector a cinco números enteros.

[6] Con más precisión, este método requiere un argumento *iterador*, no un argumento índice en número entero. Los algoritmos `begin()` y `end()` devuelven iteradores, a los cuales se les pueden aplicar desplazamientos. En esto, son parecidos a los apuntadores (que se presentan en el siguiente capítulo).

A continuación, se usa la instrucción `nums_partes.push_back(157);` para añadir el número entero 157 al final del vector, lo cual produce los siguientes elementos:

```
136  142  122  109  144  157
```

Por último, la sección final del código usado en el programa 11.11 usa el algoritmo `sort()` para ordenar los elementos en el vector. Después que se aplica el algoritmo, se despliegan de nuevo los valores en el vector. A continuación se presenta la salida completa producida por el programa 11.11:

```
El vector inicialmente tiene un tamanio de 4,
 y contiene los elementos:
136   122   109   146

Después de reemplazar el cuarto elemento, el vector tiene un
 tamanio de 4, y contiene los elementos:
136  122  109  144

Después de insertar un elemento en la segunda posición,
 el vector tiene un tamanio de 5, y contiene los elementos:
136  142  122  109  144

Después de agregar un elemento al final de la lista,
  el vector tiene un tamanio de 6, y contiene los elementos:
136  142  122  109  144  157

Después de ordenarlos, los elementos del vector son:
109  122  136  142  144  157
```

Ejercicios 11.6

1. Defina los términos “contenedor” y “Biblioteca Estándar de Plantillas”.
2. ¿Qué instrucciones `include` deberían incluirse con programas que usan la Biblioteca Estándar de Plantillas?
3. Introduzca y ejecute el programa 11.11.
4. Modifique el programa 11.11 de modo que el usuario introduzca el conjunto de números inicial cuando se ejecute el programa. Haga que el programa solicite la cantidad de números iniciales que se van a introducir.
5. Modifique el programa 11.11 para usar y desplegar los resultados reportados por los métodos `capacity ()` y `max_size()` de la clase de vector.
6. Modifique el programa 11.11 para usar el algoritmo `random_shuffle()`.
7. Modifique el programa 11.11 para usar los algoritmos `binary_search()` y `find()`. Haga que su programa solicite el número que se va a encontrar.
8. Usando el programa 11.11 como punto de partida, cree un programa equivalente que use un vector de cadenas. Inicialice el vector usando el arreglo `string nombres[] = {"Donovan", "Michaels", "Smith", "Jones"};`.

9. Use los algoritmos `max_element()` y `min_element()` para determinar los valores máximo y mínimo, respectivamente, en el vector creado para el ejercicio 8. (*Sugerencia*: Use la expresión `max_element(nombreVector.begin(), nombreVector.end())` para determinar el valor máximo almacenado en el vector. Luego use los mismos argumentos para el algoritmo `min_element()`.)

10. Vuelva a hacer la aplicación 2 en la sección 11.4 usando un vector en lugar de un arreglo.

11.7 ERRORES COMUNES DE PROGRAMACIÓN

Cuatro errores comunes se asocian con el uso de arreglos.

1. Olvidar declarar el arreglo. Este error produce un mensaje de error del compilador equivalente a "indirección inválida" cada vez que se encuentre una variable subindexada dentro de un programa. El significado exacto de este mensaje de error se aclarará cuando se establezca la correspondencia entre arreglos y apuntadores en el capítulo 12.
2. Usar un subíndice que haga referencia a un elemento del arreglo inexistente. Por ejemplo, declarar que el arreglo es de tamaño 20 y usar un valor subíndice de 25. Este error no es detectado por la mayor parte de los compiladores de C++. Sin embargo, producirá un error en tiempo de ejecución que causará una caída del programa o que se tenga acceso en la memoria a un valor que no tiene relación con el elemento pretendido. En cualquier caso, por lo general es un error difícil en extremo de localizar. La única solución para este problema es asegurarse, ya sea por instrucciones de programación específicas o por una codificación cuidadosa, que cada subíndice hace referencia a un elemento del arreglo válido.
3. No usar un valor contador lo bastante grande en un contador de ciclo `for` para recorrer todos los elementos del arreglo. Este error ocurre por lo general cuando en un principio se especifica que un arreglo es de tamaño *n* y hay un ciclo `for` dentro del programa de la forma `for (i = 0; i < n; i++)`. El tamaño del arreglo es expandido luego, pero el programador olvida cambiar los parámetros del ciclo for interior. Declarar un tamaño de arreglo usando una constante nombrada y usar de manera consistente la constante nombrada a lo largo de la función en lugar de la variable *n*, la cual elimina este problema.
4. Olvidar inicializar el arreglo. Aunque muchos compiladores establecen de manera automática todos los elementos de arreglos valorados en números enteros y reales en cero, y todos los elementos de arreglos de carácter en espacios en blanco, le corresponde al programador asegurarse que cada arreglo es inicializado en forma correcta antes que comience el procesamiento de los elementos del arreglo.

11.8 RESUMEN DEL CAPÍTULO

1. Un arreglo unidimensional es una estructura de datos que puede utilizarse para almacenar una lista de valores del mismo tipo de datos. Tales arreglos deben declararse dando el tipo de datos de los valores que están almacenados en el arreglo y el tamaño del arreglo. Por ejemplo, la declaración:

   ```
   int num[100];
   ```

   crea un arreglo de 100 números enteros. Un enfoque más adecuado es usar primero una constante nombrada para establecer el tamaño del arreglo, y luego utilizar esta constante en la definición del arreglo. Por ejemplo,

   ```
   const int TAMANIOMAX = 100;
   ```

   and

   ```
   int num[TAMANIOMAX];
   ```

2. Los elementos del arreglo son almacenados en ubicaciones contiguas en la memoria y se referencian usando el nombre del arreglo y un subíndice, por ejemplo, `num[22]`. Puede utilizarse cualquier expresión de valor en número entero no negativo como subíndice y el subíndice 0 siempre se refiere al primer elemento en un arreglo.

3. Un arreglo bidimensional se declara enlistando un tamaño de fila y uno de columna con el tipo de datos y el nombre del arreglo. Por ejemplo, la declaración

   ```
   int mat[5][7];
   ```

   crea un arreglo bidimensional consistente de cinco filas y siete columnas de valores enteros.

4. Los arreglos pueden inicializarse cuando se declaran. Para arreglos bidimensionales esto se logra enlistando los valores iniciales, fila por fila, dentro de llaves y separándolos con comas. Por ejemplo, la declaración

   ```
   int vals[3][2] = { {1, 2},
                      {3, 4},
                      {5, 6} };
   ```

   produce el siguiente arreglo de 3 filas por 2 columnas:

   ```
   1  2
   3  4
   5  6
   ```

   Como C++ usa la convención de que la inicialización procede en un orden por fila, las llaves interiores pueden omitirse. Por tanto, una inicialización equivalente es proporcionada por la instrucción:

   ```
   int vals[3][2] = { 1, 2, 3, 4,  5, 6};
   ```

5. Los arreglos son transmitidos a una función pasando el nombre del arreglo como un argumento. El valor transmitido en realidad es la dirección de la primera ubicación de almacenamiento del arreglo. Por tanto, la función linvocada recibe acceso directo al arreglo original y no a una copia de los elementos del arreglo. Dentro de la función llamada debe declararse un argumento formal para que reciba el nombre del arreglo transmitido. La declaración del argumento formal puede omitir el tamaño de filas del arreglo.

11.9 APÉNDICE DEL CAPÍTULO: BÚSQUEDA Y ORDENAMIENTO

La mayoría de los programadores se encuentran con la necesidad de ordenar y buscar en una lista de elementos de datos en algún momento en sus carreras de programación. Por ejemplo, los resultados experimentales podrían tener que ser colocados en orden ascendente o descendente para su análisis estadístico, listas de nombres pueden tener que ser clasificadas en orden alfabético, o una lista de fechas puede tener que reordenarse en orden ascendente de fechas. Del mismo modo, puede tener que buscar en una lista de nombres para encontrar un nombre particular en la lista, o puede tener que buscar en una lista de fechas para localizar una fecha particular. En esta sección se introducen los fundamentos del ordenamiento y búsqueda en listas. hay que observar que no es necesario ordenar una lista antes de hacer una búsqueda aunque, como se verá, son posibles búsquedas más rápidas si la lista está ordenada.

Algoritmos de búsqueda

Un requerimiento común de muchos programas es buscar un elemento determinado en una lista. Por ejemplo, en una lista de nombres y números telefónicos, podría buscarse un nombre específico de modo que pueda imprimirse el número telefónico correspondiente, o podría desearse buscar en la lista tan sólo para determinar si un nombre está ahí. Los dos métodos más comunes para llevar a cabo dichas búsquedas son los algoritmos de búsqueda lineal y binaria.

Búsqueda lineal

En una **búsqueda lineal**, la cual también se conoce como **búsqueda secuencial**, cada elemento en la lista es examinado en el orden en el que ocurre en la lista, hasta que se encuentra el elemento deseado o se llega al final de la lista. Esto es análogo a ver cada nombre en el directorio telefónico, comenzando con Aardvark, Aaron, hasta que se encuentra el que se busca o hasta que se llega a Zzxgy, Zora. Es obvio que ésta no es la forma más eficiente de buscar en una lista alfabetizada larga. Sin embargo, una búsqueda lineal tiene estas ventajas:

1. El algoritmo es simple.
2. La lista no necesita estar en ningún orden particular.

En una búsqueda lineal ésta comienza en el primer elemento en la lista y continúa de manera secuencial, elemento por elemento, a lo largo de la lista. El seudocódigo para una función que ejecute una búsqueda lineal es

```
Para todos los elementos en la lista
   Compare el elemento con el elemento deseado
```

```
   Si el elemento se encuentra
      Devolver el valor de índice del elemento actual
   Termina el si
Termina el para
Devolver -1 si el elemento no se encontró
```

Hay que observar que el valor `return` de la función indica si el elemento se encontró o no. Si el valor `return` es `-1`, el elemento no estaba en la lista; de lo contrario, el valor `return` dentro del ciclo `for` proporciona el índice de la ubicación del elemento dentro de la lista.

La función `busquedaLineal()` ilustra este procedimiento como una función de C++:

```
// esta funcion devuelve la ubicación de clave en la lista
// se devuelve un -1 si no se encuentra el valor
int busquedaLineal(int lista[], int tamanio, int clave)
{
  int i;

  for (i = 0; i < tamanio; i++)
  {
    if (lista[i] == clave)
      return i;
  }

  return -1;
}
```

Al revisar `busquedaLineal()` observe que el ciclo for tan sólo se utiliza para tener acceso a cada elemento en la lista, desde el primer elemento al último, hasta que se encuentra una coincidencia con el elemento deseado. Si se localiza el elemento deseado se devuelve el valor índice del elemento actual, lo cual causa que termine el ciclo; de lo contrario, la búsqueda continúa hasta que se encuentra el final de la lista.

Para probar esta función se ha escrito una función controladora `main()` para llamar y desplegar los resultados devueltos por `busquedaLineal()`. El programa de prueba completo se ilustra en el programa 11.12.

A continuación se presentan ejecuciones de muestra del programa 11.12:

```
Introduzca el elemento que esta buscando: 101
El elemento fue encontrado en la posición índice 9
```

and

```
Introduzca el elemento que esta buscando: 65
El elemento no se encontró en la lista
```

Como ya se ha señalado, una ventaja de las búsquedas lineales es que la lista no tiene que ordenarse para ejecutar la búsqueda. Otra ventaja es que si el elemento deseado está hacia el principio de la lista, sólo se hará un pequeño número de comparaciones. El peor caso, por supuesto, ocurre cuando el elemento deseado está al final de la lista. En promedio, sin embargo, y suponiendo que el elemento deseado tiene una probabilidad igual de estar en cualquier parte dentro de la lista, el número de comparaciones requeridas será $n/2$, donde n es el tamaño de la lista. Por tanto, para una lista de 10 elementos, el número promedio de comparaciones necesarias para una búsqueda lineal es 5, y para una lista de 10 000 elementos, el número

promedio de comparaciones necesarias es 5 000. Como se muestra a continuación, este número puede reducirse de manera significativa usando un algoritmo de búsqueda binaria.

Programa 11.12

```
#include <iostream>
using namespace std;

int busquedaLineal(int [], int, int);

int main()
{
  const int NUMEL = 10;
  int nums[NUMEL] = {5,10,22,32,45,67,73,98,99,101};
  int elemento, ubicación;


  cout << "Introduzca el elemento que esta buscando: ";
  cin >> elemento;

  ubicación = busquedaLineal(nums, NUMEL, elemento);

  if (ubicación > -1)
    cout << "El elemento fue encontrado en la posición índice " << ubicacion
         << endl;
  else
    cout << "El elemento no se encontro en la lista\n";

  return 0;
}

// esta función devuelve la ubicación de clave en la lista
// se devuelve un -1 si no se encuentra el valor
int busquedaLineal(int lista[], int tamanho, int clave)
{
  int i;

  for (i = 0; i < tamanio; i++)
  {
    if (lista[i] == clave)
      return i;
  }

  return -1;
}
```

Búsqueda binaria

En una **búsqueda binaria** la lista debe estar ordenada. Empezando por una lista ordenada, el elemento deseado se compara primero con los elementos en medio de la lista (para listas con un número par de elementos, puede usarse cualquiera de los dos elementos intermedios). Se presentan tres posibilidades una vez que se hace la comparación: el elemento deseado puede ser igual al elemento intermedio, puede ser mayor o puede ser menor que el elemento intermedio.

En el primer caso la búsqueda ha sido exitosa, y ya no se requiere una mayor búsqueda. En el segundo caso, dado que el elemento deseado es mayor que el elemento intermedio, si se encuentra en absoluto debe estar en la segunda mitad de la lista. Esto significa que la primera parte de la lista, consistente en todos los elementos desde el primero hasta el intermedio, puede descartarse para cualquier búsqueda posterior. En el tercer caso, dado que el elemento deseado es menor que el elemento intermedio, si se encuentra debe hallarse en la primera parte de la lista. Para este caso la segunda mitad de la lista que contiene todos los elementos desde el intermedio hasta el último pueden desecharse para cualquier búsqueda posterior.

El algoritmo para implementar esta estrategia de búsqueda se ilustra en la figura 11.14 y se define en el siguiente seudocódigo:

```
Establecer el índice inferior en 0
Establecer el índice superior en uno menos que el tamaño de la lista
Comenzar con el primer elemento en la lista
Mientras el índice inferior sea menor que o igual al índice superior
   Establecer el índice del punto medio como el promedio en número entero de los valo-
   res de índice inferior y superior
   Comparar el elemento deseado con el elemento intermedio
      Si el elemento deseado es igual al elemento intermedio
         Devolver el valor del índice del elemento actual
    De no ser así si el elemento deseado es mayor que el elemento intermedio
       Establecer el valor del índice inferior en el valor del punto intermedio más 1
    De no ser así si el elemento deseado es menor que el elemento intermedio
       Establecer el valor del índice superior en el valor del punto intermedio menos 1
    Termina el si
Termina el mientras
Devolver -1 si el elemento no se encuentra
```

Como se ilustra tanto con el seudocódigo como con el diagrama de flujo de la figura 11.14, se utiliza un ciclo `while` para controlar la búsqueda. La lista inicial se define estableciendo el valor del índice izquierdo en 0 y el valor del índice derecho en uno menos que el número de elementos en la lista. El elemento intermedio se toma entonces como el promedio, en número entero, de los valores izquierdo y derecho. Una vez que se ha hecho la comparación con el elemento intermedio, la búsqueda se restringe en lo subsiguiente al mover el índice izquierdo a un valor entero por encima del punto medio o al mover el índice derecho a un valor entero por debajo del punto medio. Este proceso continúa hasta que se encuentra el elemento deseado o los valores de índice izquierdo y derecho se vuelven iguales. La función `busquedaBinaria()` presenta la versión en C++ del algoritmo.

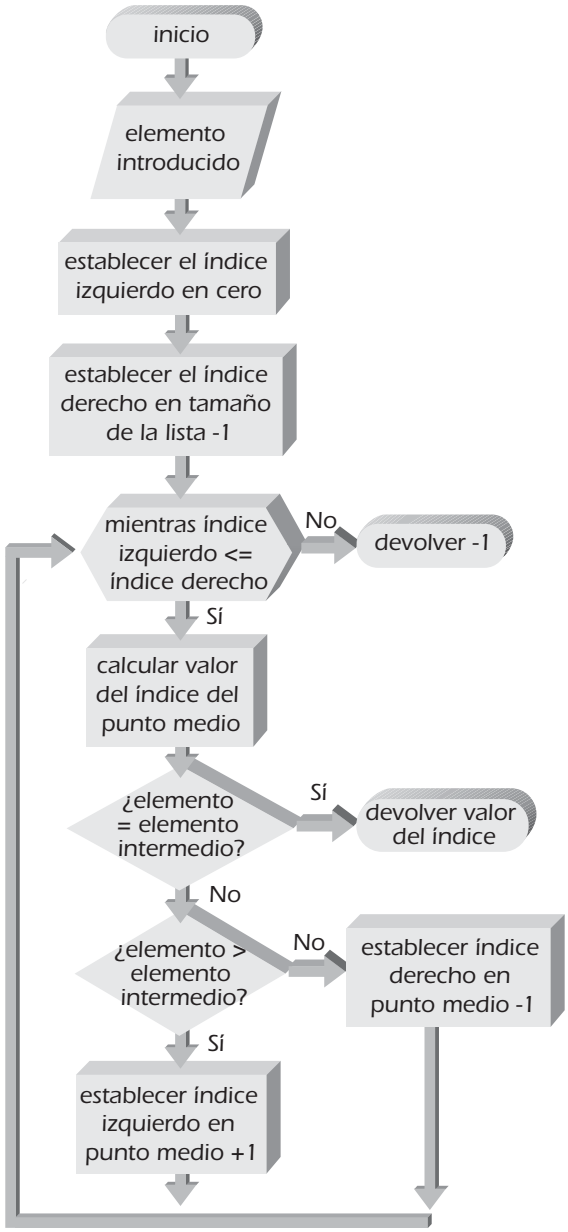


Figura 11.14 El algoritmo de búsqueda binaria.

```
// esta función devuelve la ubicación de clave en la lista
// se devuelve -1 si no se encuentra el valor
int busquedaBinaria(int lista[], int tamaño, int clave)
{
  int izquierdo, derecho, puntomedio;

  izquierdo = 0;
  derecho = tamanho -1;

  while (izquierdo <= derecho)
  {
    puntomedio = (int) ((izquierdo + derecho) / 2);
    if (clave == lista[puntomedio])
    {
      return puntomedio;
    }
    else if (clave > lista[puntomedio])
      izquierdo = puntomedio + 1;
    else
      derecho = puntomedio -1;
  }

  return -1;
}
```

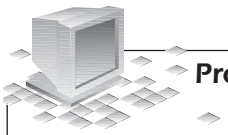
Para probar esta función se usó el programa 11.13.

A continuación se presenta una ejecución de muestra del programa 11.13:

```
Introduzca el elemento que esta buscando: 101
El elemento fue encontrado en la posición índice 9
```

La ventaja de usar un algoritmo de búsqueda binaria es que el número de elementos en el que debe buscarse se reduce a la mitad cada vez por medio del ciclo `while`. Por tanto, la primera vez se recorre el ciclo debe buscarse en n elementos; la segunda vez que se recorre el ciclo se han eliminado $n/2$ de los elementos y sólo quedan $n/2$. La tercera vez que se recorre el ciclo otra mitad de los elementos restantes se ha eliminado, y así en forma sucesiva.

En general, después de p recorridos por el ciclo, el número de valores restantes en los que se va a buscar es $n/(2^p)$. En el peor caso la búsqueda puede continuar hasta que haya menos que o igual a 1 elemento restante en el cual buscar. Matemáticamente, esto puede expresarse como $n/(2^p) \leq 1$. De manera alternativa, esto puede replantearse como que p es el número entero menor tal que $2^p \geq n$. Por ejemplo, para un arreglo de 1000 elementos, n es 1000 y el número máximo de recorridos, p, requerido para una búsqueda binaria es 10. La tabla 11.3 compara el número de recorridos del ciclo necesarios para una búsqueda lineal y binaria para varios tamaños de lista.



Programa 11.13

```
#include <iostream>
using namespace std;

int busquedaBinaria(int [], int, int);

int main()
{
  const int NUMEL = 10;
  int nums[NUMEL] = {5,10,22,32,45,67,73,98,99,101};
  int elemento, ubicación;
  cout << "Introduzca el elemento que esta buscando: ";
  cin >> elemento;
  ubicacion = busquedaBinaria(nums, NUMEL, elemento);
  if (ubicacion > -1)
    cout << "El elemento fue encontrado en la posición indice "
       << ubicacion << endl;
  else
    cout << "El elemento no se encontró en el arreglo\n";

  return 0;
}

// esta función devuelve la ubicación de clave en la lista
// se devuelve un -1 si no se encuentra el valor
int busquedaBinaria(int lista[], int tamanio, int clave)
{
  int izquierdo, derecho, puntomedio

  izquierdo = 0;
  derecho = tamanio -1;

  while (izquierdo <= derecho)
  {
    puntomedio = (int) ((izquierdo + derecho) / 2);
    if (clave == lista[puntomedio])
    {
      return puntomedio;
    }
    else if (clave > lista[puntomedio])
      izquierdo = puntomedio + 1;
    else
      derecho = puntomedio -1;
  }

  return -1;
}
```

Tabla 11.3 Una comparación de recorridos del ciclo `while` para búsquedas lineales y binarias

Tamaño del arreglo	10	50	500	5000	50 000	500 000	5 000 000	50 000 000
Recorridos de búsqueda lineal promedio	5	25	250	2500	25 000	250 000	2 500 000	25 000 000
Recorridos de búsqueda lineal máximos	10	50	500	5000	50 000	500 000	5 000 000	50 000 000
Recorridos de búsqueda binaria máximos	4	6	9	13	16	19	23	26

Como se ilustra, el número máximo de recorridos del ciclo para una lista de 50 elementos es casi 10 veces mayor para una búsqueda lineal que para una búsqueda binaria, y es aún más espectacular para listas más grandes. Como regla general, por lo común se toman 50 elementos como el punto de intercambio: para listas más pequeñas que 50 elementos son aceptables las búsquedas lineales; para listas más grandes deberá usarse un algoritmo de búsqueda binaria.

La notación de la O grande

En promedio, en una gran cantidad de búsquedas lineales con n elementos en una lista, se esperaría examinar la mitad ($n/2$) de los elementos antes de localizar el elemento deseado. En una búsqueda binaria el número máximo de recorridos, p, ocurre cuando $n/(2)^p = 1$. Esta relación puede manipularse algebraicamente a $2^p = n$, lo cual produce $p = \log_2 n$, lo cual aproximadamente es igual a $3.33\log_{10} n$.

Por ejemplo, encontrar un nombre particular en un directorio alfabético con $n = 1000$ nombres requeriría un promedio de 500 ($=n/2$) comparaciones usando una búsqueda lineal. Con una búsqueda binaria, sólo se requerirían alrededor de 10 ($\approx 3.33 * \log_{10}1000$) comparaciones.

Una forma común de expresar el número de comparaciones requeridas en cualquier algoritmo de búsqueda usando una lista de n elementos es dar el orden de magnitud del número de comparaciones requeridas, en promedio, para localizar un elemento deseado. Por tanto, se dice que la búsqueda lineal es del orden n y la búsqueda binaria del orden $\log_2 n$. Desde el punto de vista de la notación, esto se expresa como $O(n)$ y $O(\log_2 n)$, donde la O se lee como "del orden de".

Algoritmos de ordenamiento

Para ordenar datos, existen dos categorías principales de técnicas de ordenamiento, llamadas ordenamientos internos y externos, respectivamente. Los **ordenamientos internos** se usan cuando la lista de datos no es demasiado grande y la lista completa puede ordenarse dentro de la memoria de la computadora, por lo general en un arreglo. Los **ordenamientos externos** se usan para conjuntos de datos mucho más grandes que se almacenan en archivos de disco o cinta externos grandes y no pueden acomodarse dentro de la memoria de la computadora como una unidad completa. Aquí se presentan dos algoritmos de ordenamiento interno que pueden utilizarse en forma efectiva cuando se ordenan listas con menos de aproximadamente 50 elementos. Para listas más grandes de manera típica se emplean algoritmos de ordenamiento más complejos.

Ordenamiento por selección

Una de las técnicas de ordenamiento más simples es el **ordenamiento por selección**. En un ordenamiento por selección el valor más pequeño se selecciona al inicio de la lista completa

de datos y se intercambia con el primer elemento en la lista. Después de esta primera selección e intercambio, el siguiente elemento más pequeño en la lista revisada se selecciona e intercambia con el segundo elemento en la lista. Dado que el elemento más pequeño ya está en la primera posición en la lista, este segundo recorrido sólo necesita considerar del segundo al último elementos. Para una lista consistente en n elementos, este proceso se repite $n - 1$ veces, con cada recorrido a lo largo de la lista requiriendo una comparación menos que el recorrido anterior.

Por ejemplo, considérese la lista de números ilustrada en la figura 11.15. El primer recorrido a través de la lista inicial produce que se seleccione el número 32 y se intercambie con el primer elemento en la lista. El segundo recorrido, hecho en la lista reordenada, produce que se seleccione el número 155 del segundo al quinto elementos. Este valor se intercambia entonces con el segundo elemento en la lista. El tercer recorrido selecciona el número 307 del tercero al quinto elementos en la lista e intercambia este valor con el tercer elemento. Por último, el cuarto y último recorrido a través de la lista selecciona el valor mínimo restante y lo intercambia con el cuarto elemento de la lista. Aunque cada recorrido en este ejemplo produjo un intercambio, no se habría hecho ningún intercambio en un recorrido si el valor más pequeño ya estuviera en la ubicación correcta.

lista inicial	recorrido 1	recorrido 2	recorrido 3	recorrido 4
690	32	32	32	32
307	307	155	144	144
32	690	690	307	307
155	155	307	690	426
426	426	426	426	690

Figura 11.15 Una muestra de ordenamiento por selección.

En seudocódigo, el ordenamiento por selección se describe como

```
Establecer contador de intercambio en cero (no se requiere, pero se hace para dar seguimiento
a los intercambios)
Para cada elemento en la lista del primero al penúltimo
  Encontrar el elemento menor a partir del elemento actual referido al último elemento;
    Establecer el valor mínimo igual al elemento actual
    Guardar (almacenar) el índice del elemento actual
    Para cada elemento en la lista desde el elemento actual + 1 hasta el último elemento en la lista
      Si elemento[índice de ciclo interior] < valor mínimo
        Establecer valor mínimo = elemento[índice de ciclo interior]
        Guardar el índice del nuevo valor mínimo encontrado
      Termina el si
    Termina el para
    Intercambiar el valor actual con el nuevo valor mínimo
    Incrementar el contador de intercambio
Termina el para
Devolver la cuenta de intercambio
```

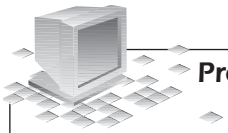
La función `ordenSeleccion()` incorpora este procedimiento en una función de C++.

```
int ordenSeleccion(int num[], int numel)
{
  int i, j, min, indicemin, temp, movimientos = 0;

  for ( i = 0; i < (numel - 1); i++)
  {
    min = num[i];  // supone que el mínimo es el primer elemento
                      del arreglo
    indicemin = i; // índice del elemento mínimo
    for(j = i + 1; j < numel; j++)
    {
      if (num[j] < min)    // si se ha localizado un valor inferior
      {                    // se captura
      min = num[j];
      indicemin = j;
      }
    }
    if (min < num[i])  // comprueba si se tiene un nuevo mínimo
    {                  // y si se tiene, intercambia valores
      temp = num[i];
      num[i] = min;
      num[indicemin] = temp;
      movimientos++;
    }
  }

  return movimientos;
}
```

La función `ordenSeleccion()` espera dos argumentos: la lista que se va a ordenar y el número de elementos en la lista. Como lo especifica el seudocódigo, un conjunto de ciclos `for` anidados ejecuta el ordenamiento. El ciclo `for` exterior causa un recorrido menos a lo largo de la lista que el número total de elementos de datos en la lista. Para cada recorrido, al principio se le asigna a la variable `min` el valor `num[i]`, donde `i` es la variable contadora del ciclo `for` exterior. dado que `i` comienza en 0 y termina en uno menos que `numel`, cada elemento en la lista, excepto el último, es designado en forma sucesiva como el elemento actual.

El ciclo interior recorre los elementos por debajo del elemento actual y se usa para seleccionar el siguiente valor más pequeño. Por tanto, este ciclo comienza en el valor índice `i + 1` y continúa hasta el final de la lista. Cuando se encuentra un nuevo mínimo, se almacenan su valor y posición en la lista en las variables llamadas `min` e `indicemin`, respectivamente. Al completar el ciclo interior se hace un intercambio sólo si se encontró un valor menor que el de la posición actual.

Con el propósito de probar `ordenSeleccion()` se construyó el programa 11.14. Este programa implementa un ordenamiento por selección para la misma lista de 10 números que se usó antes para probar los algoritmos de búsqueda. Para una comparación posterior con los otros algoritmos de ordenamiento que se presentarán, se cuenta y despliega el número de movimientos que realice el programa para ordenar los datos.



Programa 11.14

```
#include <iostream>
using namespace std;

int ordenSeleccion(int [], int);

int main()
{
  const int NUMEL = 10;
  int nums[NUMEL] = {22,5,67,98,45,32,101,99,73,10};
  int i, movimientos;

  movimientos = ordenSeleccion(nums, NUMEL);

  cout << "La lista ordenada, en orden ascendente, es:\n";
  for (i = 0; i < NUMEL; i++)
    cout << "  " <<nums[i];

  cout << "\nSe hicieron " << movimientos << " movimientos para ordenar
          esta lista\n";

  return 0;
}

int ordenSeleccion(int num[], int numel)
{
  int i, j, min, indicemin, temp, movimientos = 0;

  for ( i = 0; i < (numel - 1); i++)
  {
    min = num[i];    // supone que el mínimo es el primer elemento del arreglo
    indicemin = i;  // índice del elemento mínimo
    for(j = i + 1; j < numel; j++)
    {
      if (num[j] < min)   // si se ha localizado un valor inferior
      {                   // se captura
      min = num[j];
      indicemin = j;
      }
    }
    if (min < num[i])   // comprueba si se tiene un nuevo minimo
    {                   // y si se tiene, intercambia valores
      temp = num[i];
      num[i] = min;
      num[indicemin] = temp;
      movimientos++;
    }
  }

  return movimientos;
}
```

La salida producida por el programa 11.14 es la siguiente:

```
La lista ordenada, en orden ascendente, es:
  5  10  22  32  45  67  73  98  99  101
Se hicieron 8 movimientos para ordenar esta lista
```

Es evidente que el número de movimientos desplegado depende del orden inicial de los valores en la lista. Una ventaja del ordenamiento por selección es que el número máximo de movimientos que deben hacerse es $n - 1$, donde n es el número de elementos en la lista. Además, cada movimiento es un movimiento final que produce un elemento que reside en su ubicación final en la lista ordenada.

Una desventaja del ordenamiento por selección es que siempre se requieren $n(n-1)/2$ comparaciones, sin importar el orden inicial de los datos. Este número de comparaciones se obtiene como sigue: el último recorrido siempre requiere una comparación, el penúltimo recorrido requiere dos comparaciones, etc., hasta el primer recorrido, el cual requiere $n - 1$ comparaciones. Por tanto, el número total de comparaciones es

$$1 + 2 + 3 + \cdots + n - 1 = n(n-1)/2 = n^2/2 - n/2.$$

Para valores grandes de n domina el n^2, y el orden del ordenamiento por selección es $O(n^2)$.

Ordenamiento por intercambio (burbuja)

En un **ordenamiento por intercambio**, elementos adyacentes de la lista son intercambiados entre sí, de tal manera que la lista queda ordenada. Un ejemplo de dicha secuencia de intercambios es proporcionado por el ordenamiento burbuja, donde se comparan valores sucesivos en la lista, comenzando por los primeros dos elementos. Si la lista se va a poner en orden ascendente (de menor a mayor), el valor más pequeño de los dos que se están comparando siempre se coloca antes del valor más grande. Para listas en orden descendente (de mayor a menor), el menor de los dos valores que se están comparando siempre se coloca después del valor mayor.

Por ejemplo, suponiendo que se va a ordenar una lista de valores en orden ascendente, si el primer elemento en la lista es mayor que el segundo, los dos elementos son intercambiados. Luego se comparan el segundo y tercer elementos. Una vez más, si el segundo elemento es mayor que el tercero, estos dos elementos se intercambian. Este proceso continúa hasta que se han comparado e intercambiado, si es necesario, los últimos dos elementos. Si no se hicieron intercambios durante este recorrido inicial a lo largo de los datos, éstos están en el orden correcto y el proceso termina; de lo contrario, se hace un segundo recorrido a lo largo de los datos, comenzando desde el primer elemento y deteniéndose en el penúltimo elemento. La razón para detenerse en el penúltimo elemento en el segundo recorrido es que el primer recorrido siempre produce que el valor más positivo se "hunda" hasta el final de la lista.

Como un ejemplo específico de este proceso, considérese la lista de números ilustrada en la figura 11.16. La primera comparación produce el intercambio de los valores de los primeros dos elementos, 690 y 307. La siguiente comparación, entre los elementos dos y tres en la lista revisada, produce el intercambio de valores entre el segundo y tercer elementos, 690 y 32. Esta comparación y el posible intercambio de valores adyacentes continúa hasta que se han comparado e intercambiado, de ser el caso, los últimos dos elementos. Este proceso completa el primer recorrido a lo largo de los datos y produce que el número mayor se mueva hasta el final de la lista. Conforme el valor mayor se hunde hasta su lugar de reposo al final de la lista, los elementos más pequeños se elevan lentamente, como "burbujas", hasta el principio de la lista. Este efecto de burbuja de los elementos más pequeños es lo que le da el nombre de ordenamiento de "burbuja" a este algoritmo de ordenamiento.

690 ←	307	307	307	307
307 ←	690 ←	32	32	32
32	32 ←	690 ←	155	155
155	155	155 ←	690 ←	426
426	426	426	426 ←	690

Figura 11.16 El primer recorrido de un ordenamiento por intercambio.

Debido a que el primer recorrido a lo largo de la lista asegura que el valor mayor siempre se mueve al final de la lista, el segundo recorrido se detiene en el penúltimo elemento. Este proceso continúa con cada recorrido deteniéndose en un elemento anterior que el recorrido previo, hasta que se han completado $n - 1$ recorridos a lo largo de la lista o no son necesarios intercambios en cualquier recorrido. En ambos casos la lista resultante está ordenada. El seudocódigo que describe este ordenamiento es

```
Establecer contador de intercambio en cero (no se requiere, pero se hace para dar seguimiento a los intercambios)
Para el primer elemento en la lista hasta llegar a uno menor que el último elemento (índice i)
  Para el segundo elemento en la lista hasta el último elemento (índice j)
    Si num[j] < num[j – 1]
    {
      Intercambiar num[j] con num[j – 1]
      Incrementar contador de intercambio
    }
  Termina el para
Termina el para
Devolver cuenta de intercambio
```

Este algoritmo de ordenamiento se codifica en C++ como la función `ordenBurbuja()`, la cual se incluye dentro del programa 11.15 con propósitos de prueba. Este programa prueba `ordenBurbuja()` con la misma lista de 10 números usada en el programa 11.14 para probar `ordenSeleccion()`. Para hacer una comparación con el ordenamiento por selección anterior, también se cuenta y despliega el número de movimientos adyacentes (intercambios) hechos por `ordenBurbuja()`.

Aquí está la salida producida por el programa 11.15.

```
La lista ordenada, en orden ascendente, es:
  5  10  22  32  45  67  73  98  99  101
Se hicieron 18 movimientos para ordenar esta lista
```

Como con el ordenamiento por selección, el número de comparaciones usando un ordenamiento de burbuja es $O(n^2)$ y el número de movimientos requerido depende del orden inicial de los valores en la lista. En el peor caso, cuando los datos están en orden inverso, el ordenamiento por selección se desempeña mejor que el ordenamiento de burbuja. Aquí ambos ordenamientos requieren $n(n - 1)/2$ comparaciones, pero el ordenamiento por selección sólo necesita $n - 1$ movimientos mientras el ordenamiento de burbuja necesita $n(n - 1)/2$ movimientos. Los movimientos adicionales requeridos por el ordenamiento de burbuja resultan de los intercambios intermedios entre elementos adyacentes para "colocar" cada elemento en su posición final. En este aspecto el ordenamiento por selección es superior, porque no son necesarios movimientos intermedios. Para datos aleatorios, como los usados en los programas 11.14 y 11.15, el ordenamiento por selección por lo general se desempeña igual o mejor que el ordenamiento de burbuja.

Programa 11.15

```
#include <iostream>
using namespace std;

int ordenBurbuja(int [], int);

int main()
{
  const int NUMEL = 10;
  int nums[NUMEL] = {22,5,67,98,45,32,101,99,73,10};
  int i, movimientos;

  movimientos = ordenBurbuja(nums, NUMEL);

  cout << "La lista ordenada, en orden ascendente, es:\n";
  for (i = 0; i < NUMEL; ++i)
    cout << "   " <<nums[i];

  cout << "\nSe hicieron " << movimientos << " movimientos para ordenar
          esta lista\n";

  return 0;
}

int ordenBurbuja(int num[], int numel)
{
  int i, j, temp, movimientos = 0;

  for ( i = 0; i < (numel - 1); i++)
  {
    for(j = 1; j < numel; j++)
    {
      if (num[j] < num[j-1])
      {
        temp = num[j];
        num[j] = num[j-1];
        num[j-1] = temp;
        movimientos++;
      }
    }
  }

  return movimientos;
}
```

CAPÍTULO 12

Apuntadores

TEMAS

12.1 DIRECCIONES Y APUNTADORES
ALMACENAMIENTO DE DIRECCIONES — DECLARACIÓN DE APUNTADORES
USO DE DIRECCIONES — REFERENCIAS Y APUNTADORES

12.2 NOMBRES DE ARREGLOS COMO APUNTADORES
ASIGNACIÓN DINÁMICA DE ARREGLOS

12.3 ARITMÉTICA DE APUNTADORES
INICIALIZACIÓN DE APUNTADORES

12.4 TRANSMISIÓN DE DIRECCIONES
TRANSMISIÓN DE ARREGLOS
NOTACIÓN AVANZADA PARA APUNTADORES

12.5 ERRORES COMUNES DE PROGRAMACIÓN

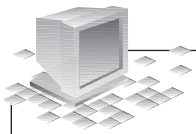
12.6 RESUMEN DEL CAPÍTULO

Una de las ventajas de C++ es que permite al programador tener acceso a las direcciones de variables usadas en un programa. Esto permite a los programadores entrar en forma directa en el funcionamiento interno de una computadora y tener acceso a su estructura de almacenamiento básica; lo que da al programador en C++ capacidades y poder de programación que no está disponible en otros lenguajes de nivel alto. Esto se logra usando una característica llamada apuntadores. Aunque otros lenguajes proporcionan apuntadores, C++ extiende esta característica al proveer aritmética de apuntadores; es decir, los valores de apuntadores pueden sumarse, restarse y compararse.

De manera fundamental, los apuntadores son tan sólo variables que se usan para almacenar direcciones de memoria. Este capítulo presenta lo esencial de la declaración de apuntadores, y luego proporciona métodos de aplicación de variables apuntadoras para tener acceso a sus direcciones almacenadas y usarlas en formas significativas.

12.1 DIRECCIONES Y APUNTADORES

Como se vio en la sección 2.5, para desplegar la dirección de una variable se puede usar el operador de dirección de C++, `&`. Para determinar la dirección de `num` puede usarse el operador de dirección de C++, `&`, el cual significa "la dirección de". Cuando se utiliza en una instrucción que no es declarativa, el operador de dirección antepuesto al nombre de una variable se refiere a la dirección de la variable.[1] Por ejemplo, en una instrucción que no es declarativa, `&num` significa *la dirección* de `num`, `millas` significa *la dirección* de `millas` y `&foo` significa la dirección de `foo`. El programa 12.1, el cual es una copia del programa 2.10, usa el operador de dirección para desplegar la dirección de la variable `num`.



Programa 12.1

```
#include <iostream>
using namespace std;

int main()
{
  int num;

  num = 22;
  cout << "num = " << num << endl;
  cout << "La dirección de num = " << &num << endl;

  return 0;
}
```

La salida del programa 12.1 es

```
num = 22
La dirección de num = 0012FED4
```

La figura 12.1 ilustra tanto el contenido como la dirección de la variable `num` proporcionada por la salida del programa 12.1.

[1] Como se vio en el capítulo 6, cuando se usa para declarar argumentos de referencia, el signo & se refiere al tipo de datos que lo *precede*. Por tanto, tanto la declaración `double& num;` como `double #` se leen como "`num` es la dirección de un `double`" o, de manera más común, "`num` es una referencia a un `double`".

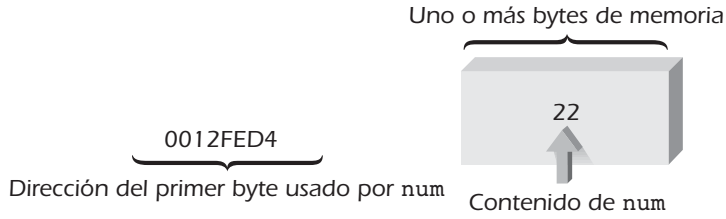


Figura 12.1 *Una representación más completa de la variable* `num`.

Como se mencionó en la sección 2.5, la información de dirección cambiará dependiendo de cuál computadora esté ejecutando el programa y cuántos otros programas estén cargados en la memoria en ese momento.

Almacenamiento de direcciones

Además de desplegar la dirección de una variable, como se hizo en el programa 12.1, también se puede almacenar direcciones en variables declaradas de manera adecuada. Por ejemplo, la instrucción

```
dirNum = &num;
```

almacena la dirección correspondiente a la variable `num` en la variable `dirNum`, como se ilustra en la figura 12.2. Del mismo modo, las instrucciones

```
d = &m;
apunta_tab = &lista;
apunta_car = &car;
```

almacenan las direcciones de las variables `m`, `lista` y `car` en las variables `d`, `apunta_tab` y `apunta_car`, respectivamente, como se ilustra en la figura 12.3.

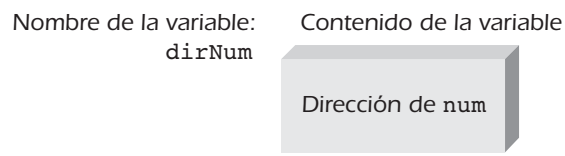


Figura 12.2 *Almacenamiento de la dirección de* `num` *en* `dirNum`.

Las variables `dirNum, d, apunta_tab` y `apunta_car` se llaman de manera formal ***variables apuntadoras***, o ***apuntadores*** para abreviar. Los ***apuntadores*** son tan sólo variables que se usan para almacenar las direcciones de otras variables.

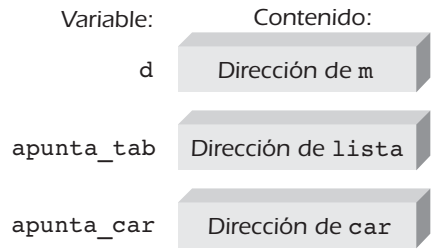


Figura 12.3 *Almacenamiento de más direcciones.*

Uso de direcciones

Para usar una dirección almacenada, C++ proporciona un **operador de indirección**, *. El símbolo *, cuando es seguido por un apuntador (con un espacio permitido tanto antes como después de *), significa *la variable cuya dirección está almacenada en*. Por tanto, si `dirNum` es un apuntador (recuérdese que un apuntador es una variable que almacena una dirección), `*dirNum` significa *la variable cuya dirección está almacenada en* `dirNum`. Del mismo modo, `*apunta_tab` significa *la variable cuya dirección está almacenada en* `apunta_tab` y `*apunta_car` significa *la variable cuya dirección está almacenada en* `apunta_car`. La figura 12.4 muestra la relación entre la dirección contenida en una variable apuntadora y la variable direccionada finalmente.

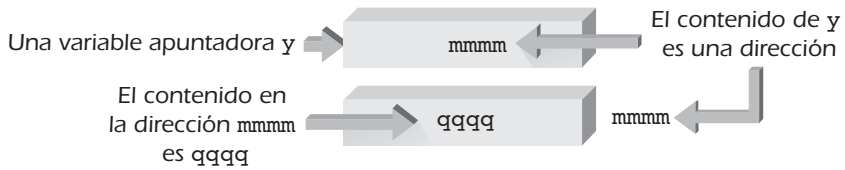


Figura 12.4 *Uso de una variable apuntadora.*

Aunque `*d` significa de manera literal *la variable cuya dirección está almacenada en* `d`, esto por lo común se abrevia como *la variable a la que apunta* `d`. Del mismo modo, refiriéndose a la figura 12.4, `*y` puede leerse como *la variable a la que apunta* `y`. El valor obtenido al final, como se muestra en la figura 12.4, es `qqqq`.

Cuando se usa una variable apuntadora, el valor que se obtiene al final siempre se encuentra yendo primero a la variable apuntadora (o apuntador, para abreviar) por una dirección. La dirección contenida en el apuntador se usa entonces para obtener el contenido deseado. Desde luego, ésta es una forma bastante indirecta de conseguir el valor final y, como era de esperar, se usa el término direccionamiento indirecto para describir este procedimiento.

Debido a que el uso de un apuntador requiere que la computadora haga una búsqueda doble (primero se recupera la dirección, luego se usa la dirección para recuperar los datos reales), una pregunta que vale la pena es, ¿por qué desearía almacenar una dirección en primer lugar? La respuesta a esta pregunta se encuentra en la relación íntima entre apuntadores y arreglos y en la capacidad de los apuntadores para crear y eliminar en forma dinámica ubicaciones de almacenamiento de variables nuevas, mientras un programa se está ejecutando. Estos temas se presentan más adelante en este capítulo. Por ahora, sin

embargo, dado que cada variable tiene una dirección de memoria asociada con ella, la idea de almacenar en verdad una dirección no debería parecer demasiado extraña.

Declaración de apuntadores

Como todas las variables, los apuntadores deben ser declarados antes que puedan utilizarse para almacenar una dirección. Cuando se declara una variable apuntadora, C++ requiere que también se especifique el tipo de variable al que apunta. Por ejemplo, si la dirección en el apuntador `dirNum` es la dirección de un número entero, la declaración correcta para el apuntador es

```
int *dirNum;
```

Esta declaración se lee como *la variable a la que apunta* `dirNum` (de `*dirNum` en la declaración) *es un número entero*.[2]

Hay que observar que la declaración int `*dirNum`; especifica dos cosas: primera, que la variable a la que apunta `dirNum` es un número entero, y segunda, que `dirNum` debe ser un apuntador (debido a que se usa con el operador de indirección *). Del mismo modo, si el apuntador `apunta_tab` apunta a (contiene la dirección de) un número de precisión doble y `apunta_car` apunta a una variable de carácter, las declaraciones requeridas para estos apuntadores son

```
double *tabPoint;
char *chrPoint;
```

Estas dos declaraciones se pueden leer, respectivamente, como la variable a la que apunta `tabPoint` es una doble variable señalada por `chrPoint` es un `char`. Porque aparecen todas las direcciones iguales, esta información adicional es necesitada por el recopilador para saber cuántas localizaciones del almacenaje al acceso cuando utiliza la dirección almacenada en el indicador. Otros ejemplos de las declaraciones del indicador son

```
char *enClave;
int *pt_num;
double *Imprnum1;
```

Para entender las declaraciones de apuntadores, es útil leerlas "a la inversa", empezando con el operador de indirección, el asterisco, `*`, y traduciéndolo como *la variable cuya dirección está almacenada en* o *como la variable a la que apunta*. Al aplicar esto a declaraciones de apuntadores, la declaración `char *enClave`, por ejemplo, puede leerse como la variable cuya dirección está almacenada en `enClave` es un carácter o como la variable a la que apunta `enClave` *es un carácter*. Ambos enunciados se abrevian con frecuencia en el enunciado más simple `enClave` *apunta* a *un carácter*. Debido a que las tres interpretaciones de la instrucción de declaración son correctas, puede seleccionarse y usarse cualquier descripción que tenga más sentido para usted. Ahora uniremos todo esto para construir un programa usando apuntadores. Considérese el programa 12.2.

---

[2]Las declaraciones de apuntadores también pueden escribirse en la forma *tipoDatos* nombreApuntador*; donde se coloca un espacio entre el símbolo del operador de indirección y el nombre de la variable apuntadora. Sin embargo, esta forma es propensa a error cuando se declaran múltiples variables apuntadoras en la misma instrucción de declaración y el símbolo de asterisco se omite de manera inadvertida después que se declara el nombre del primer apuntador. Por ejemplo, la declaración `int* num1, num2;` declara `num1` como una variable apuntadora y `num2` como una variable en número entero. Con el fin de acomodar con más facilidad múltiples apuntadores en la misma declaración y marcar con claridad una variable como apuntador, nos apegaremos a la convención que coloca un asterisco de manera directa enfrente del nombre de cada variable apuntadora. Este posible error ocurre rara vez con declaraciones de referencia debido a que las referencias se usan casi exclusivamente como parámetros formales y es obligatoria la declaración única de parámetros.

Programa 12.2

```
#include <iostream>
using namespace std;

int main()
{
  int *dirNum;         // declara un apuntador a un int
  int millas, dist;    // declara dos variables enteras

  dist = 158;          // almacena el número 158 en dist
  millas = 22;         // almacena el número 22 en millas
  dirNum = &millas;    // almacena la 'direccion de millas' en numDir

  cout << "La dirección almacenada en dirNum es " << dirNum << endl;
  cout << "El valor al que apunta dirNum es " << *dirNum << "\n\n";

  dirNum = &dist;   // ahora almacena la dirección de dist en dirNum
  cout << "La dirección almacenada ahora en dirNum es " << dirNum << endl;
  cout << "El valor al que apunta ahora dirNum es " << *dirNum << endl;

  return 0;
}
```

La salida del programa 12.2 es:

```
La dirección almacenada en dirNum es 0012FEC8
El valor al que apunta dirNum es 22

La dirección almacenada ahora en dirNum es 0012FEBC
El valor al que apunta ahora dirNum es 158
```

El único uso del programa 12.12 es ayudar a entender "qué está almacenado y dónde". Se revisará el programa para ver cómo se produjo la salida.

La instrucción de declaración int `*dirNum`; declara que `dirNum` es una variable apuntadora usada para almacenar la dirección de una variable en número entero. La instrucción `dirNum = &millas;` almacena la dirección de la variable millas en el apuntador `dirNum`. La primera activación de `cout` causa que se despliegue esta dirección. La segunda activación de `cout` en el programa 12.2 usa el operador de indirección para recuperar e imprimir *el valor al que apunta* `dirNUm`, el cual es, por supuesto, el valor almacenado en millas.

Debido a que `dirNum` se ha declarado como un apuntador a una variable en número entero, se puede usar este apuntador para almacenar la dirección de cualquier variable en número entero. La instrucción `dirNum = &dist` ilustra esto al almacenar la dirección de la variable `dist` en `dirNum`. Las últimas dos instrucciones `cout` verifican el cambio en el valor de `dirNum` y que la nueva dirección almacenada apunta a la variable `dist`. Como se ilustra en el programa 12.2, sólo deberán almacenarse direcciones en los apuntadores.

Por supuesto que habría sido mucho más simple si el apuntador usado en el programa 12.2 se hubiera podido declarar como `pointer dirNum;`. Dicha declaración, sin embargo, no transmite información del almacenamiento usado por la variable cuya dirección está almacenada en `dirNum`. Esta información es esencial cuando se usa el apuntador con el operador de indirección, como en la segunda instrucción `cout` en el programa 12.2. Por ejemplo, si la dirección de un número entero es almacenada en `dirNum`, entonces por lo general sólo se recuperan cuatro bytes de almacenamiento cuando se usa la dirección. Si la dirección de un carácter se almacena en `dirNum`, sólo se recuperaría un byte de almacenamiento, y un `double` por lo general requiere la recuperación de ocho bytes de almacenamiento. La declaración de un apuntador debe incluir, por consiguiente, el tipo de variable a la que se apunta. La figura 12.5 ilustra este concepto.

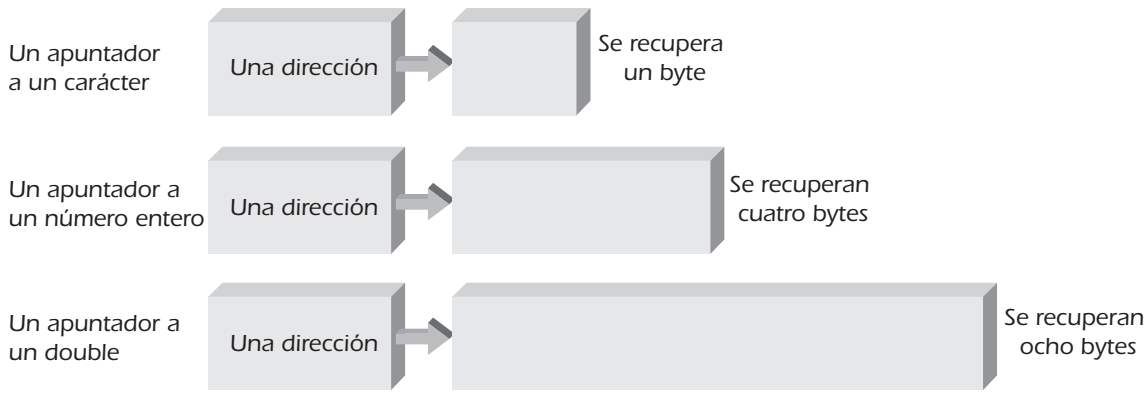


Figura 12.5 *Direccionamiento de diferentes tipos de datos usando apuntadores.*

Referencias y apuntadores

En este punto podría preguntarse cuál es la diferencia entre un apuntador y una referencia. En esencia, una referencia es una constante nombrada para una dirección; como tal, la dirección nombrada como una referencia no puede ser alterada. Dado que un apuntador es una variable, la dirección en el apuntador puede cambiarse. Para la mayor parte de las aplicaciones el uso de referencias sobre los apuntadores como argumentos para funciones es más fácil y se prefiere con claridad. Esto se debe a la notación más simple usada para localizar un parámetro de referencia, la cual elimina el uso del operador de dirección, `&`, y el operador para desreferenciar, `*`, requerido por los apuntadores. Desde el punto de vista técnico, se dice que las referencias se ***desreferencian automáticamente*** o se ***desreferencian implícitamente*** (los dos términos se usan como sinónimos), mientras los apuntadores deben desreferenciarse de manera explícita para localizar el valor al que tienen acceso.

Por ejemplo, al transmitir la dirección de una variable escalar como un argumento de una función, las referencias proporcionan una interfaz de notación más simple y por lo general se prefieren. Los apuntadores se requieren para otras situaciones, como asignar de manera dinámica secciones de memoria nuevas para variables adicionales mientras se ejecuta un programa o cuando se usan alternativas a la notación de arreglos (ambos temas se presentan en este capítulo).

Variables de referencia[3]

Las referencias se usan de manera casi exclusiva como parámetros formales de función y como tipos a devolver. No obstante, las variables de referencia también están disponibles en C++. Para completar el tema, ahora se muestra cómo pueden declararse y usarse estas variables.

Una vez que se ha declarado una variable se le pueden asignar nombres adicionales. Esto se logra usando una declaración de referencia, la cual tiene la forma

tipoDatos& nuevoNombre = nombreExistente;

Por ejemplo, la declaración de referencia

```
double& suma = total;
```

iguala el nombre `suma` al nombre `total`, ambos se refieren ahora a la misma variable, como se ilustra en la figura 12.6.

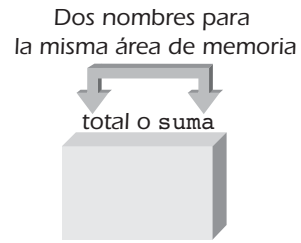


Figura 12.6 `suma` es un nombre alternativo para `total`.

Una vez que se ha establecido otro nombre para una variable usando una declaración de referencia, el nuevo nombre, el cual se conoce como un alias, puede usarse en lugar del nombre original. Por ejemplo, considérese el programa 12.3.

La siguiente salida es producida por el programa 12.3:

```
suma = 20.5
total = 18.6
```

Dado que la variable `suma` tan sólo es otra referencia a la variable `total`, es el valor almacenado en total el que es obtenido por el primer objeto `cout` en el programa 12.3. Cambiar el valor en `suma` cambia entonces el valor en `total`, el cual es desplegado por el segundo objeto `cout` en el programa 12.3.

[3]Esta sección puede omitirse sin perder la continuidad temática.

Programa 12.3

```
#include <iostream>
using namespace std;

int main()
{
  double total = 20.5     // declara e inicializa total
  double& suma = total;   // declara otro nombre para total

  cout << "suma = " << suma << endl;
  suma = 18.6;            // esto cambia el valor en total
  cout << "total = " << total << endl;

  return 0;
}
```

Al construir referencias, deben tenerse en cuenta dos consideraciones. Primera, la referencia deberá ser del mismo tipo de datos que la variable a la que se refiere. Por ejemplo, la secuencia de declaraciones

```
int num = 5;
double& numref = num;    // INVALIDO, CAUSA UN ERROR DE
                            COMPILADOR
```

no iguala `numref` con `num`; más bien, causa un error de compilador porque las dos variables son de tipos de datos diferentes. En segundo lugar, se produce un error de compilador cuando se hace un intento para igualar una referencia a una constante. Por ejemplo, la siguiente declaración es inválida:

```
int& val = 5; // INVALIDO, CAUSA UN ERROR DE COMPILADOR
```

Una vez que se ha igualado en forma correcta un nombre de referencia con un nombre de variable, la referencia no puede cambiarse para referirse a otra variable.

Como con todas las instrucciones de declaración, pueden declararse múltiples referencias en una sola instrucción en tanto cada nombre de referencia esté precedido por el símbolo &. Por tanto, la declaración

```
double& suma = total, & promedio;
```

crea dos variables de referencia llamadas suma y promedio.[4]

[4]Las declaraciones de referencia también pueden escribirse en la forma *tipoDatos &nombreNuevo = nombreExistente*; donde se coloca un espacio entre el símbolo & y el tipo de datos. Sin embargo, esta forma no se usa mucho, tal vez para distinguir la notación de dirección de una variable de referencia de la usada al asignar direcciones a las variables apuntadoras.

Otra forma de considerar las referencias es tomarlas como apuntadores con capacidades restringidas que ocultan de manera implícita una gran cantidad de desreferenciación que se requiere de manera explícita con los apuntadores.

Por ejemplo, considérense las instrucciones

```
int b;       // b es una variable en número entero
int& a = b; // a es una variable de referencia que almacena la
               direccion de b
a = 10;      // esto cambia el valor de b a 10
```

Aquí, `a` es declarada como una variable de referencia que efectivamente es una constante nombrada para la dirección de la variable `b`. Dado que el compilador sabe a partir de la declaración que `a` es una variable de referencia, de manera automática asigna la dirección de `b` (en lugar del contenido de `b`) a `a` en la instrucción de declaración. Por último, en la instrucción `a = 10;` el compilador usa la dirección almacenada en `a` para cambiar el valor almacenado en `b` a 10. La ventaja de usar la referencia es que ejecuta de manera automática un acceso indirecto al valor de `b` sin la necesidad de usar de forma explícita el símbolo de indirección, `*`. Como se ha señalado antes, este tipo de acceso se conoce como ***desreferencia automática***.

Para implementar esta misma correspondencia entre `a` y `b` usando apuntadores se hace con la siguiente secuencia de instrucciones:

```
int b;       // b es una variable en número entero
int *a = &b; // a es un apuntador, almacena la direccion
                de b en a
*a = 10;     // esto cambia el valor de b a 10 al
                desreferenciarlo
             // de manera explícita de la direccion en a
```

Aquí `a` se define como un apuntador que se ha inicializado para almacenar la dirección de `b`. Por tanto, `*a`, lo cual puede leerse como "la variable cuya dirección está en `a`" o "la variable a la que apunta `a`" es `b`, y la expresión `*a = 10` cambia el valor de `b` a `10`. Hay que observar en el caso del apuntador que la dirección almacenada puede alterarse para apuntar a otra variable; en el caso de la referencia la variable de referencia no puede alterarse para referirse a cualquier variable excepto aquella con la que se inicializó. También hay que observar que para desreferenciar `a` se debe usar de manera explícita el operador de indirección, `*`. Como podría esperarse, el símbolo `*` también se conoce como operador de desreferencia.

Ejercicios 12.1

1. Si `promedio` es una variable, ¿qué significa `&promedio`?
2. Para las variables y direcciones ilustradas en la figura 12.7, determine `&temp`, `&dist`, `&fecha` y `&millas`.

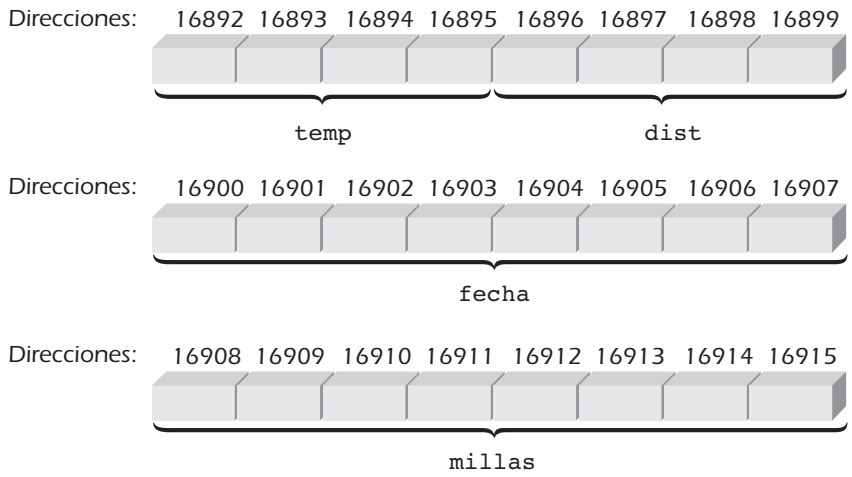


Figura 12.7 *Bytes de memoria para el ejercicio 2.*

3. a. Escriba un programa en C++ que incluya las siguientes instrucciones de declaración. Haga que el programa use el operador de dirección y el objeto `cout` para desplegar las direcciones correspondientes a cada variable.

```
int num, cuenta;
long fecha;
float pendiente;
double potencia;
```

b. Después de ejecutar el programa escrito para el ejercicio 3a, dibuje un diagrama de la manera en que su computadora ha apartado almacenamiento para las variables en el programa. En su diagrama, llene las direcciones desplegadas por el programa.

c. Modifique el programa escrito en el ejercicio 3a para desplegar la cantidad de almacenamiento que reserva su computadora para cada tipo de datos (use el operador `sizeof()`). Con esta información y la información de direcciones proporcionada en el ejercicio 3b, determine si su computadora apartó almacenamiento para las variables en el orden en que fueron declaradas.

4. Si una variable es declarada como un apuntador, ¿qué debe almacenarse en la variable?

5. Usando el operador de indirección, escriba expresiones para lo siguiente:

a. La variable a la que apunta `dir_x`
b. La variable cuya dirección está en `dir_y`
c. La variable a la que apunta `pt_prod`
d. La variable a la que apunta `pt_millas`
e. La variable a la que apunta `impr_m`
f. La variable cuya dirección está en `fecha_p`
g. La variable a la que apunta `impr_dist`

h. La variable a la que apunta `pt_tab`
i. La variable cuya dirección está en `pt_horas`

6. Escriba instrucciones de declaración para lo siguiente:
a. La variable a la que apunta `dir_y` es un número entero.
b. La variable a la que apunta `dir_car` es un carácter.
c. La variable a la que apunta `pt_anio` es un entero largo.
d. La variable a la que apunta `cant` es una variable de precisión doble.
e. La variable a la que apunta `z` es un número entero.
f. La variable a la que apunta `qp` es una variable de precisión simple.
g. `pt_fecha` es un apuntador a un número entero.
h. `dir_prod` es un apuntador a una variable de precisión doble.
i. `pt_cant` es un apuntador a una variable de precisión simple.
j. `pt_car` es un apuntador a un carácter.

7. **a.** ¿Cómo se denominan las variables `dir_y, dir_car, pt_anio, cant, z, qp, pt_fecha, dir_prod, pt_cant` y `pt_car` usadas en el ejercicio 6?
b. ¿Por qué los nombres de las variables `cant`, `z` y `qp` usados en el ejercicio 6 no son buenas elecciones para nombres de variables apuntadoras?

8. Escriba enunciados en español que describan lo que contienen las siguientes variables declaradas:
a. `char *dir_clave;`
b. `int *m;`
c. `double *dir_prod;`
d. `long *impr_y;`
e. `double *pCou;`
f. `int *pt_fecha;`

9. ¿Cuáles de las siguientes son declaraciones para apuntadores?
a. `long a;`
b. `char b;`
c. `char *c;`
d. `int x;`
e. `int *p;`
f. `double w;`
g. `float *k;`
h. `float l;`
i. `double *z;`

10. Para las siguientes declaraciones,

```
int *pt_x, *dir_y;
long *dir_dt, *dir_pt;
double *pt_Z;
int a;
long b;
double c;
```

determine cuál de las siguientes instrucciones es válida:

```
a. dir_y = &a;     b. dir_y = &b;       c. dir_y = &c;
d. dir_y = a;      e. dir_y = b;        f. dir_y = c;
g. dir_dt = &a;    h. dir_dt = &b;      i. dir_dt = &c;
j. dir_dt = a;     k. dir_dt = b;       l. dir_dt = c;
m.pt_z = &a;       n. dir_pt = &b;      o. dir_pt = &c;
p. dir_pt = a;     q. dir_pt = b;       r. dir_pt = c;
s. dir_y = pt_x;   t. dir_y = dir_dt;   u. dir_y = dir_pt;
```

11. Para las variables y direcciones ilustradas en la figura 12.8, escriba los datos apropiados determinados por las siguientes instrucciones:

```
a. pt_num = &m;
b. dir_cant = &cant;
c. *dir_z = 25;
d. k = *dir_num;
e. pt_dia = dir_z;
f. *pt_anio = 1987;
g. *dir_cant = *dir_num;
```

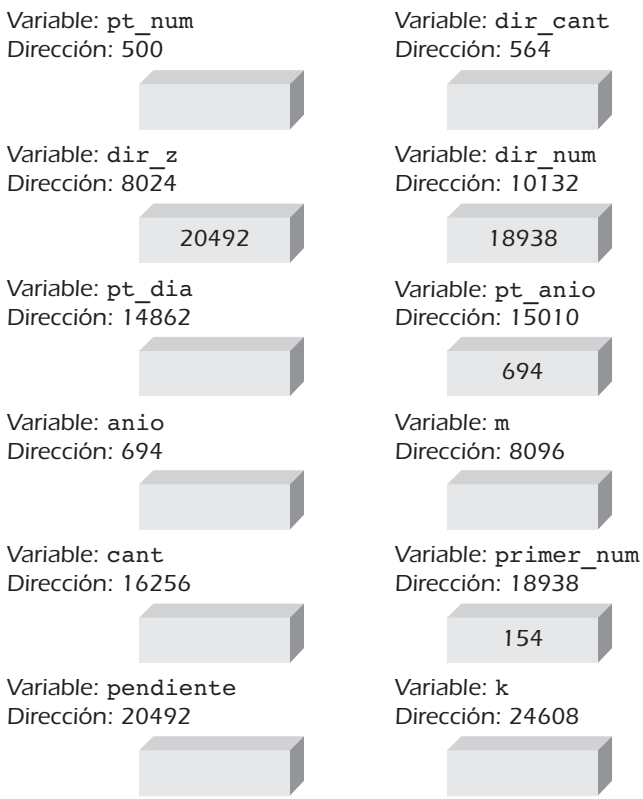


Figura 12.8 Ubicaciones de memoria para el ejercicio 11.

12. Usando el operador `sizeof()`, determine el número de bytes usados por su computadora para almacenar la dirección de un número entero, un carácter y un número de precisión doble. (*Sugerencia:* puede utilizar `sizeof(*int)` para determinar el número de bytes de memoria usados por un apuntador a un número entero.) ¿Esperaría que fuera igual el tamaño de todas las direcciones? ¿Por qué sí o por qué no?

12.2 NOMBRES DE ARREGLOS COMO APUNTADORES

Aunque los apuntadores son, por definición, tan sólo variables usadas para almacenar direcciones, también existe una relación directa y estrecha entre los nombres de los arreglos y los apuntadores. En esta sección se describe esta relación con detalle.

La figura 12.9 ilustra el almacenamiento de un arreglo unidimensional llamado `calif`, el cual contiene cinco números enteros. Suponga que cada número entero requiere dos bytes de almacenamiento.

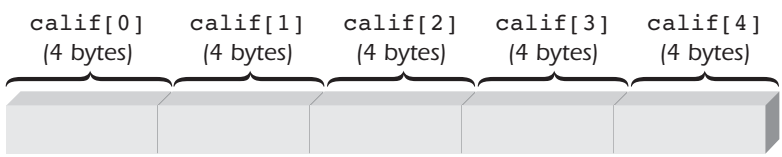


Figura 12.9 *El arreglo* `calif` *en almacenamiento.*

Usando subíndices, se hace referencia al cuarto elemento en el arreglo calif como `calif[3]`. El uso de un subíndice, sin embargo, oculta el uso extenso de direcciones por la computadora. Interiormente, la computadora utiliza de inmediato el subíndice para calcular la dirección del elemento deseado basándose tanto en la dirección inicial del arreglo como en la cantidad de almacenamiento usado por cada elemento. Llamar al cuarto elemento `calif[3]` obliga al compilador, en forma interna, a hacer el cálculo de dirección

```
&calif[3] = &calif[0] + (3 * sizeof(int))
```

Si se recuerda que el operador de dirección, `&`, significa "la dirección de", esta última instrucción se lee "la dirección de `calif[3]` es igual a la dirección de `calif[0]` más 12". La figura 12.10 ilustra el cálculo de dirección usado para localizar `calif[3]`.

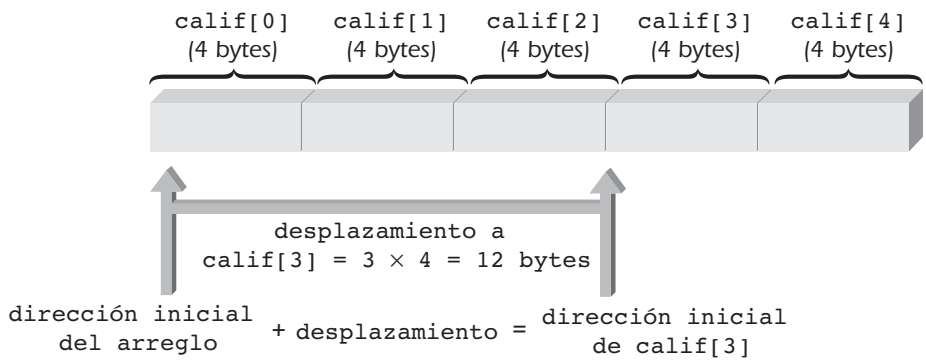
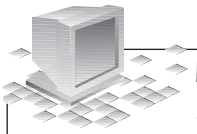


Figura 12.10 *Uso de un subíndice para obtener una dirección.*

Hay que recordar que un apuntador es una variable usada para almacenar una dirección. Si se crea un apuntador para almacenar la dirección del primer elemento en el arreglo `calif`, se puede imitar la operación utilizada por la computadora para tener acceso a los elementos del arreglo. Antes de hacer esto, se considerará primero el programa 12.4.

Cuando se ejecuta el programa 12.4, se obtiene el siguiente despliegue:

```
El elemento 0 es 98
El elemento 1 es 87
El elemento 2 es 92
El elemento 3 es 79
El elemento 4 es 85
```



Programa 12.4

```
#include <iostream>
using namespace std;

int main()
{
  const int TAMANIOARREGLO = 5;

  int i, calif[TAMANIOARREGLO] = {98, 87, 92, 79, 85};

  for (i = 0; i < TAMANIOARREGLO; i++)
    cout << "\nEl elemento " << i << " es " << calif[i];
  cout << endl;

  return 0;
}
```

El programa 12.4 despliega los valores del arreglo `calif` usando notación de subíndice estándar. Ahora, se almacenará la dirección del elemento 0 del arreglo en un apuntador. Entonces, usando el operador de indirección, `*`, se puede usar la dirección en el apuntador para tener acceso a cada elemento del arreglo. Por ejemplo, si se almacena la dirección de `calif[0]` en un apuntador llamado `g_Ptr` (usando la instrucción de asignación `g_Ptr = &calif[0];`), entonces, como se ilustra en la figura 12.11, la expresión `*g_Ptr`, la cual significa "la variable a la que apunta `g_Ptr`", hace referencia a `calif[0]`.

Una característica única de los apuntadores es que pueden incluirse desplazamientos en las expresiones que usan apuntadores. Por ejemplo, el 1 en la expresión `*(g_Ptr + 1)` es un **desplazamiento**. La expresión completa hace referencia al número entero que está un lugar adelante de la variable a la que apunta `g_Ptr`. Del mismo modo, como se ilustra en la figura 12.12, la expresión `*(g_Ptr + 3)` hace referencia a la variable que está tres números enteros adelante de la variable a la que apunta `g_Ptr`. Ésta es la variable `calif[3]`.

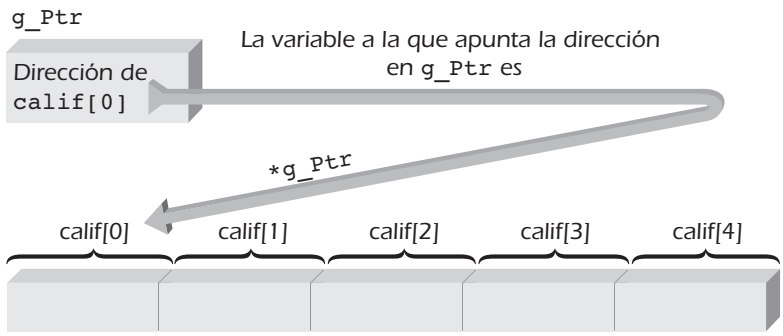


Figura 12.11 *La variable a la que apunta* `*g_Ptr` *es* `calif[0]`.

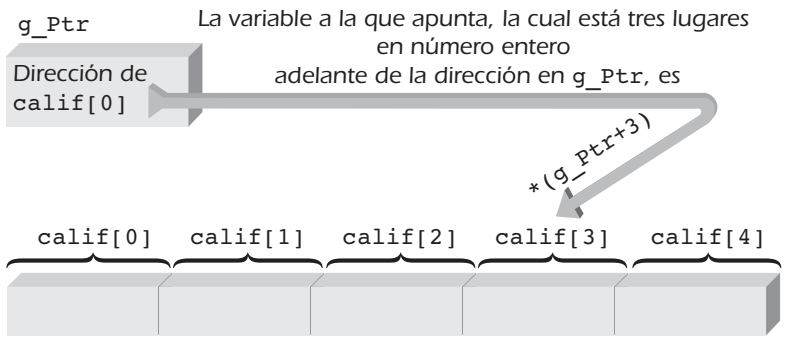


Figura 12.12 *Un desplazamiento de tres desde la dirección en* `g_Ptr`.

La tabla 12.1 muestra la correspondencia completa entre elementos referenciados por subíndices y por apuntadores y desplazamientos. Las relaciones enumeradas en la tabla 12.1 se ilustran en la figura 12.13.

Tabla 12.1 Los elementos del arreglo se pueden referenciar de dos maneras

Elemento del arreglo	Notación de subíndice	Notación de apuntador
Elemento 0	`calif[0]`	`*g_Ptr y (g_Ptr + 0)`
Elemento 1	`calif[1]`	`*(g_Ptr + 1)`
Elemento 2	`calif[2]`	`*(g_Ptr + 2)`
Elemento 3	`calif[3]`	`*(g_Ptr + 3)`
Elemento 4	`calif[4]`	`*(g_Ptr + 4)`

Usando la correspondencia entre apuntadores y subíndices ilustrada en la figura 12.13, ahora pueden utilizarse apuntadores para tener acceso a los elementos del arreglo a los que se tuvo acceso antes en el programa 12.4 usando subíndices. Esto se hace en el programa 12.5.

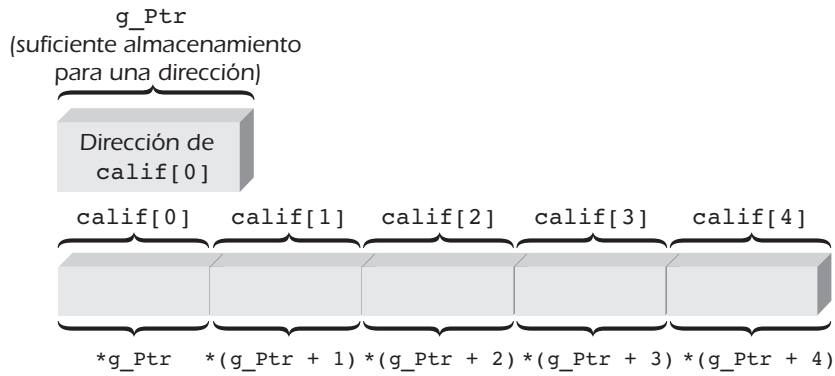


Figura 12.13 *La relación entre elementos del arreglo y apuntadores.*

Programa 12.5

```
#include <iostream>
using namespace std;

int main()
{

  const int TAMANIOARREGLO = 5;

  int *g_Ptr;                // declara un apuntador a un int
  int i, calif[TAMANIOARREGLO] = {98, 87, 92, 79, 85};

  g_Ptr = &cali[0];         // almacena la dirección inicial del arreglo
  for (i = 0; i < TAMANIOARREGLO; i++)
    cout << "\nEl elemento " << i << " es " << *(g_Ptr + i);
  cout << endl;

  return 0;
}
```

Cuando se ejecuta el programa 12.5 se obtiene el siguiente despliegue:

```
El elemento 0 es 98
El elemento 1 es 87
El elemento 2 es 92
El elemento 3 es 79
El elemento 4 es 85
```

Hay que observar que es el mismo despliegue producido por el programa 12.4.

El método usado en el programa 12.5 para tener acceso a elementos individuales del arreglo simula la forma en que el compilador hace referencia de manera interna a todos los elementos del arreglo. Cualquier subíndice usado por un programador es convertido de manera automática a una expresión de apuntador equivalente por el compilador. En este caso, dado que la declaración de `g_Ptr` incluía la información de que se apunta a números enteros, cualquier desplazamiento añadido a la dirección en `g_Ptr` se escala de manera automática al tamaño de un número entero. Por tanto, `*(g_Ptr + 3)`, por ejemplo, se refiere a la dirección de `calif[0]` más un desplazamiento de doce bytes `(3 * 4)`, donde se ha hecho la suposición que `sizeof(int) = 4`. Ésta es la dirección de `calif[3]` ilustrada en la figura 12.13.

Los paréntesis en la expresión `*(g_Ptr + 3)` son necesarios para hacer referencia en forma correcta al elemento del arreglo deseado. Omitir los paréntesis produce la expresión `*g_Ptr + 3`. Debido a la precedencia de los operadores, esta expresión agrega 3 a "la variable a la que apunta `g_Ptr`". Dado que `g_Ptr` apunta a `calif[0]`, esta expresión suma el valor de `calif[0]` y 3. Hay que observar también que la expresión `*(g_Ptr + 3)` no cambia la dirección almacenada en `g_Ptr`. Una vez que la computadora usa el desplazamiento para localizar la variable correcta a partir de la dirección inicial en `g_Ptr`, el desplazamiento es desechado y la dirección en `g_Ptr` permanece sin cambios.

Aunque el apuntador `g_Ptr` usado en el programa 12.4 se creó de manera específica para almacenar la dirección inicial del arreglo `calif`, esto era, de hecho, innecesario. Cuando se crea un arreglo, el compilador crea de manera automática una constante apuntadora interna para éste y almacena la dirección inicial del arreglo en este apuntador. En casi todos los aspectos, una constante apuntadora es idéntica a una variable apuntadora creada por un programador; pero, como se verá, hay algunas diferencias.

Para cada arreglo creado, el nombre del arreglo se convierte en el nombre de la constante apuntadora creada por el compilador para el arreglo, y la dirección inicial de la primera ubicación reservada para el arreglo es almacenada en este apuntador. Por tanto, al declarar el arreglo `calif` en los programas 12.4 y 12.5 en realidad se reservó suficiente almacenamiento para cinco números enteros, se creó un apuntador interno llamado `calif` y almacenó la dirección de `calif[0]` en el apuntador. Esto se ilustra en la figura 12.14.

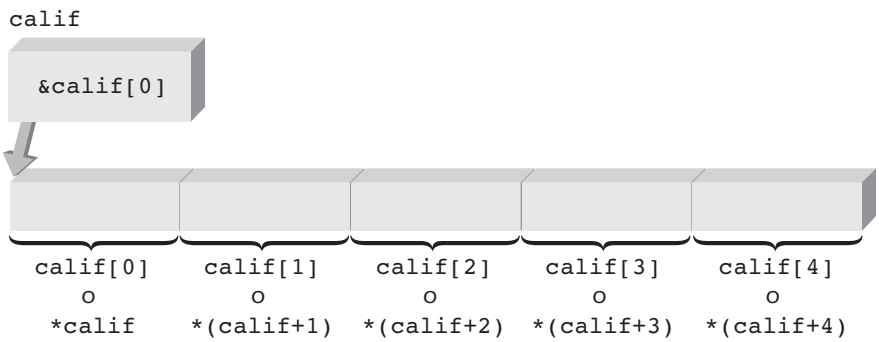
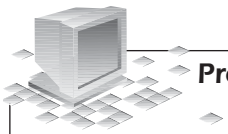


Figura 12.14 *Crear un arreglo también crea un apuntador.*

La implicación es que toda referencia a `calif` utilizando un subíndice puede reemplazarse por una referencia equivalente usando `calif` como apuntador. Por tanto, siempre que se use la expresión `calif[i]`, también puede emplearse la expresión `*(calif + i)`. Esto se ilustra en el programa 12.6, donde se usa `calif` como un apuntador para hacer referencia a todos sus elementos.

La ejecución del programa 12.6 produce la misma salida realizada antes por el programa 12.4 y el programa 12.5. Sin embargo, usar `calif` como apuntador hace innecesario declarar e inicializar el apuntador `g_Ptr` utilizando en el programa 12.5.

En la mayor parte de los aspectos un nombre de arreglo y un apuntador pueden usarse de manera indistinta. *Sin embargo, un apuntador verdadero es una variable y la dirección almacenada en ella puede cambiarse. Un nombre de arreglo es una constante apuntadora y la dirección almacenada en el apuntador no puede ser cambiada por una instrucción de asignación*. Por tanto, una instrucción como `calif = &calif[2];` es inválida. Esto no debería ser una sorpresa. En vista que el propósito de un nombre de arreglo es localizar de manera correcta el principio del arreglo, permitir a un programador cambiar la dirección almacenada en el nombre del arreglo iría en contra de este propósito y conduciría a estragos siempre que se hiciera referencia a sus elementos. Además, las expresiones que toman la dirección de un nombre de arreglo son inválidas porque el apuntador creado por el compilador es interno en la computadora, y no está almacenado en la memoria, como las variables apuntadoras. Por tanto, tratar de almacenar la dirección de `calif` usando la expresión `&calif` produce un error de compilador.



Programa 12.6

```
#include <iostream>
using namespace std;

int main()
{
  const int TAMNIOARREGLO = 5;

  int i, calif[TAMANIOARREGLO] = {98, 87, 92, 79, 85};

  for (i = 0; i < TAMANIOARREGLO; i++)
    cout << "\nEl elemento " << i << " es " << *(calif + i);
  cout << endl;

  return 0;
}
```

Un aspecto secundario interesante de la observación de que puede hacerse referencia a los elementos de un arreglo usando apuntadores es que una referencia con apuntador siempre puede reemplazarse con una referencia con subíndice. Por ejemplo, si se declara `num_Ptr` como una variable apuntadora, la expresión `*(num_Ptr + i)` también puede escribirse como `num_Ptr[i]`. Esto es cierto aun cuando `num_Ptr` no es creado como un arreglo. Como antes, cuando el compilador encuentra la notación de subíndice, la reemplaza en forma interna con la notación de apuntador.

Asignación dinámica de arreglos[5]

Conforme se define cada variable en un programa se le asigna suficiente almacenamiento de una reserva de ubicaciones de memoria en la computadora que se ponen a disposición del compilador. Una vez que se han reservado ubicaciones de memoria específicas para una variable, estas ubicaciones se fijan para toda la vida de esa variable, se usen o no. Por ejemplo, si una función solicita almacenamiento para un arreglo de 500 números enteros, se asigna y se fija el almacenamiento para el arreglo desde el punto de la definición del arreglo. Si la aplicación requiere menos de 500 números enteros, el almacenamiento asignado no utilizado no es liberado de nuevo en el sistema hasta que el arreglo deja de existir. Por otra parte, si la aplicación requiere más de 500 números enteros, el tamaño del arreglo de números enteros debe incrementarse y recompilarse la función que define al arreglo.

Una alternativa para esta asignación fija o estática de ubicaciones de almacenamiento en memoria es la asignación dinámica de memoria. Bajo un esquema de ***asignación dinámica***, la cantidad de almacenamiento que se va a asignar es determinada y ajustada conforme se ejecuta el programa, en lugar de fijarse en tiempo de compilación.

La asignación dinámica de memoria es útil en extremo cuando se manejan listas, debido a que permite que se expanda la lista conforme se agregan nuevos elementos y se contraiga conforme se eliminan. Por ejemplo, al construir una lista de calificaciones puede no conocerse el número exacto de calificaciones que se van a necesitar a final de cuentas. En lugar de crear un arreglo fijo para almacenar las calificaciones, es muy útil tener un mecanismo mediante el cual el arreglo pueda agrandarse y reducirse según sea necesario. Dos operadores en C++, `new` y `delete`, que proporcionan esta capacidad se describen en la tabla 12.2. (Estos operadores requieren el archivo de encabezado `stdlib.h`).

Tabla 12.2 Los operadores `new` ***y*** `delete` ***(requieren el archivo de encabezado*** `new`***)***

Nombre del operador	Descripción
`new`	Reserva el número de bytes solicitado por la declaración. Devuelve la dirección de la primera ubicación reservada o `NULL` si no se dispone de suficiente memoria.
`delete`	Libera un bloque de bytes reservado con anterioridad. La dirección de la primera ubicación reservada debe transmitirse como un argumento al operador.

Las solicitudes de almacenamiento dinámico explícito de variables escalares o arreglos se hacen como parte de una instrucción de declaración o de asignación.[6] Por ejemplo, la instrucción de declaración `int *num = new int;` reserva un área suficiente para contener un número entero y coloca la dirección de esta área de almacenamiento en el apuntador `num`. Esta misma asignación dinámica puede hacerse declarando primero el apuntador usando la instrucción de declaración `int *num;` y luego asignar al apuntador una dirección con la instrucción de asignación `num = new int;`. En cualquier caso el área de almacenamiento asignada proviene del área de almacenamiento libre de la computadora.[7]

[5] Este tema puede omitirse en la primera lectura sin perder la continuidad temática.

[6] Debe señalarse que el compilador proporciona de manera automática esta asignación y desasignación dinámicas desde la pila para todas las variables auto.

[7] El área de almacenamiento libre de una computadora se conoce de manera formal como *montículo*. El montículo consiste en memoria no asignada que puede asignarse a un programa, según se requiera, mientras el programa se está ejecutando.

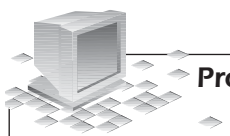
Una manera similar y de mayor utilidad es la asignación dinámica de arreglos. Por ejemplo, la declaración

```
int *calif = new int[200];
```

reserva un área suficiente para almacenar 200 números enteros y coloca la dirección del primer número entero en el apuntador `calif`. Aunque se ha usado la constante 200 en este ejemplo de declaración, puede usarse una dimensión variable. Por ejemplo, considérese la secuencia de instrucciones

```
cout << "Introduzca el número de calificaciones que se van
a procesar: ";
cin >> numcalif;
int *calif = new int[numcalif];
```

En esta secuencia el tamaño real del arreglo que se crea depende del número introducido por el usuario. Dado que los nombres del apuntador y del arreglo están relacionados, puede tenerse acceso a cada valor en el área de almacenamiento recién creada usando una notación de arreglo estándar, como `calif[i],` en lugar de la notación de apuntador equivalente `*(calif + i)`. El programa 12.7 ilustra esta secuencia de código en el contexto de un programa completo.



Programa 12.7

```
#include <iostream>
#include <new>
using namespace std;

int main()
{
  int numcalif, i;

  cout << "Introduzca el número de calificaciones que se van a procesar: ";
  cin >> numcalif;

  int *calif = new int[numcalif]; // crea el arreglo

  for (i = 0; i < numcalif; i++)
  {
    cout << " Introduzca una calificación: ";
    cin >> calif[i];
  }
  cout << "\nSe creó un arreglo para " << numcalif << " números enteros\n";
  cout << " Los valores almacenados en el arreglo son: ";
  for (i = 0; i < numcalif; i++)
    cout << "\n   " << calif[i]
  cout << endl;

  delete[] calif;     // devuelve el almacenamiento al montículo

  return 0;
}
```

En el programa 12.7 se puede observar que el operador `delete` se usa con corchetes siempre que se haya empleado antes el operador `new` para crear un arreglo. La instrucción `delete[]` reintegra al sistema operativo el bloque de almacenamiento asignado mientras el programa se está ejecutando.[8] La única dirección requerida por `delete` es la dirección inicial del bloque de almacenamiento que fue asignado en forma dinámica. Por tanto, cualquier dirección devuelta por new puede ser usada en lo subsiguiente por `delete` para reintegrar a la computadora la memoria reservada. El operador `delete` no altera la dirección que se le transmite, sino tan sólo elimina el almacenamiento al que hace referencia la dirección. A continuación hay una muestra de la ejecución del programa 12.7:

```
Introduzca el número de calificaciones que se van a proce-
sar: 4
  Introduzca una calificación: 85
  Introduzca una calificación: 96
  Introduzca una calificación: 77
  Introduzca una calificación: 92

Se creó un arreglo para 4 números enteros
 Los valores almacenados en el arreglo son:
   85
   96
   77
   92
```

Ejercicios 12.2

1. Reemplace cada una de las siguientes referencias a una variable subindexada con una referencia a un apuntador.

   a. `precios[5]` **b.** `calif[2]` **c.** `produc[10]`
   d. `dist[9]` **e.** `millas[0]` **f.** `temp[20]`
   g. `celsius[16]` **h.** `num[50]` **i.** `tiempo[12]`

2. Reemplace cada una de las siguientes referencias usando un apuntador con una referencia subindexada.

   a. `*(mensaje + 6)` **b.** `*cant` **c.** `*(anios + 10)`
   d. `*(existencias + 2)` **e.** `*(tasas + 15)` **f.** `*(codigos + 19)`

3. **a.** Enliste las tres acciones que causan que el compilador genere la instrucción de declaración `double pendientes[5];`.
   b. Si cada número de precisión doble usa ocho bytes de almacenamiento, ¿cuánto almacenamiento se reserva para el arreglo `pendientes`?
   c. Trace un diagrama parecido a la figura 12.14 para el arreglo `pendientes`.
   d. Determine el desplazamiento en bytes relativo al inicio del arreglo `pendientes`, correspondiente al desplazamiento en la expresión `*(pendientes + 3)`.

[8] El almacenamiento asignado será devuelto de manera automática al montículo, por el sistema operativo, cuando el programa haya completado su ejecución. Sin embargo, dado que esto no siempre sucede así, es importante en extremo reintegrar formalmente al montículo la memoria asignada de manera dinámica cuando el almacenamiento ya no se necesita. Se usa el término **fuga de memoria** para describir la condición que ocurre cuando la memoria asignada en forma dinámica no se devuelve formalmente usando el operador `delete` y el sistema operativo no reclama el área de memoria asignada.

4. Escriba una declaración para almacenar los siguientes valores en un arreglo llamado `tasas`: 12.9, 18.6, 11.4, 13.7, 9.5, 15.2, 17.6. Incluya la declaración en un programa que despliegue los valores en el arreglo usando notación de apuntador.

12.3 ARITMÉTICA DE APUNTADORES

Las variables apuntadoras, como todas las variables, contienen valores. El valor almacenado en un apuntador es, por supuesto, una dirección. Por tanto, al sumar y restar números a los apuntadores se pueden obtener direcciones diferentes. Además, las direcciones en los apuntadores pueden compararse usando cualquiera de los operadores relacionales (`==`, `!=`, `<` , `>`, etc.) que son válidos para comparar otras variables. Al realizar aritmética en los apuntadores debe tenerse cuidado de producir direcciones que apunten a algo significativo. Al comparar apuntadores también deben hacerse comparaciones que tengan sentido. Considérense las declaraciones:

```
int nums[100];
int *nPt;
```

Para establecer la dirección de `nums[0]` en `nPt` se puede usar cualquiera de las siguientes dos instrucciones de asignación:

```
nPt = &nums[0];
nPt = nums;
```

Las dos instrucciones de asignación producen el mismo resultado debido a que `nums` es una constante apuntadora que contiene la dirección de la primera ubicación en el arreglo. Ésta es, por supuesto, la dirección de `nums[0]`. La figura 12.15 ilustra la asignación de memoria resultante de las instrucciones de declaración y de asignación previas, suponiendo que cada número entero requiere cuatro bytes de memoria y que la ubicación del inicio del arreglo `nums` está en la dirección 18934.

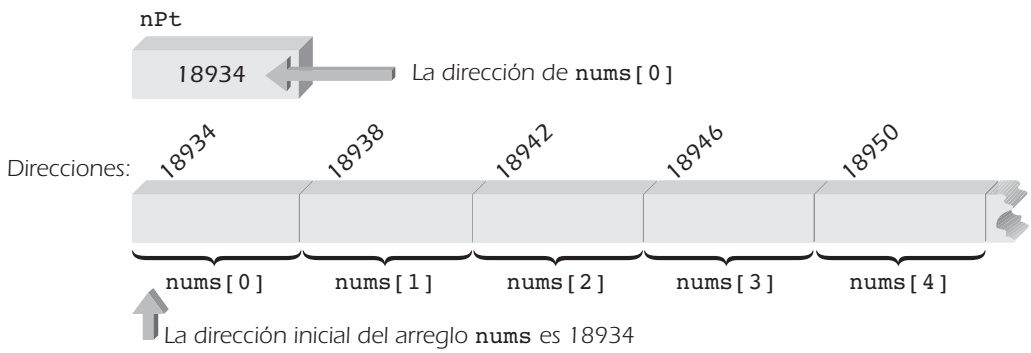


Figura 12.15 El arreglo `nums` en la memoria.

Una vez que `nPt` contiene una dirección válida, pueden sumarse y restarse valores de la dirección para producir direcciones nuevas. Cuando se suman o restan números a apuntadores, la computadora ajusta de manera automática el número para asegurar que el resultado aún "apunta" a un valor del tipo correcto. Por ejemplo, la instrucción `nPt = nPt +4;` obliga a la computadora a escalar el 4 por el número correcto para asegurar que la

dirección resultante es la dirección de un número entero. Suponiendo que cada número entero requiere cuatro bytes de almacenamiento, como si ilustró en la figura 12.15, la computadora multiplica el 4 por 4 y suma 16 a la dirección en `nPt`. La dirección resultante es 18950, la cual es la dirección correcta de `nums[4]`.

Este escalamiento automático que hace la computadora asegura que la expresión `nPt + i`, donde `i` es cualquier número entero positivo, apunte en forma correcta al `i-ésimo` elemento delante de aquel al que apunta en la actualidad `nPt`. Por tanto, si `nPt` contiene inicialmente la dirección de `nums[0]`, `nPt + 4` es la dirección de `nums[4]`, `nPt + 50` es la dirección de `nums[50]` y `nPt + i` es la dirección de `nums[i]`. Aunque se han usado direcciones reales en la figura 12.15 para ilustrar el proceso de escalamiento, el programador no necesita conocer ni preocuparse por las direcciones reales utilizadas por la computadora. La manipulación de direcciones usando apuntadores por lo general no requiere el conocimiento de la dirección real.

Las direcciones también pueden aumentarse o disminuirse usando prefijos y posfijos con operadores de incremento y decremento. Agregar uno a un apuntador causa que éste apunte al siguiente elemento del tipo al que se apunta. Disminuir un apuntador causa que éste apunte al elemento anterior. Por ejemplo, si la variable apuntadora `p` es un apuntador a un número entero, la expresión `p++` causa que la dirección en el apuntador sea incrementada para apuntar al siguiente número entero. Esto se ilustra en la figura 12.16.

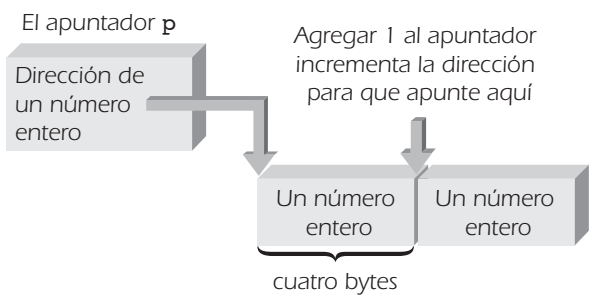


Figura 12.16 *Los incrementos se escalan cuando se usan con apuntadores.*

Al revisar la figura 12.16 se puede observar que el incremento añadido al apuntador es escalado en forma correcta para justificar el hecho que el apuntador se usa para apuntar a números enteros. Por supuesto, le corresponde al programador asegurarse que el tipo de datos correctos está almacenado en la nueva dirección contenida en el apuntador.

Los operadores de incremento y decremento pueden aplicarse como operadores de apuntador de prefijo o posfijo. Todas las siguientes combinaciones que usan apuntadores son válidas:

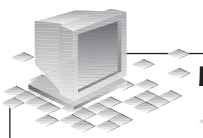
```
*ptNum++    // usa el apuntador y luego lo incrementa
*++ptNum    // incrementa el apuntador antes de usarlo
*ptNum--    // usa el apuntador y luego lo decrementa
*--ptNum    // decrementa el apuntador antes de usarlo
```

De las cuatro formas posibles, la más usada es la forma `*ptNum++`. Esto se debe a que dicha expresión permite que se tenga acceso a cada elemento en el arreglo conforme la dirección "se mueve" desde la dirección inicial del arreglo hasta la dirección del último elemento del arreglo. El uso del operador de incremento se muestra en el programa 12.8. En este programa cada elemento en el arreglo `nums` se recupera incrementando de manera sucesiva la dirección en `nPt`.

La salida producida por el programa 12.8 es:

```
El total de los elementos del arreglo es 115
```

La expresión `total = total + *nPt++` usada en el programa 12.8 acumula los valores "a los que apunta" la variable apuntadora `nPt`. Dentro de esta expresión, el término `*nPt++` causa primero que la computadora recupere el número entero al que apunta `nPt`. Esto lo realiza la parte `*nPt` del término. El posfijo de incremento, `++`, añade luego uno a la dirección en `nPt` de modo que `nPt` ahora contiene la dirección del siguiente elemento del arreglo. El incremento es escalado, por supuesto, por la computadora de modo que la dirección real en `nPt` es la dirección correcta del siguiente elemento.



Programa 12.8

```
#include <iostream>
using namespace std;

int main()
{
  const int NUMS = 5;

  int nums[NUMS] = {16, 54, 7, 43, -5};
  int i, total = 0, *nPt;

  nPt = nums;     // almacena la dirección de nums[0] en nPt
  for (i = 0; i < NUMS; i++)
    total = total + *nPt++;
  cout << "El total de los elementos del arreglo es " << total << endl;

  return 0;
}
```

Los apuntadores también pueden compararse. Esto es útil en particular cuando se trata con apuntadores que apuntan a elementos en el mismo arreglo. Por ejemplo, en lugar de usar un contador en un ciclo `for` para tener acceso en forma correcta a cada elemento en un arreglo, la dirección en un apuntador puede compararse con la dirección inicial y final del mismo arreglo. La expresión

```
nPt <= &nums[4]
```

es verdadera (no es cero) en tanto la dirección en `nPt` sea menor que o igual a la dirección de `nums[4]`. Dado que `nums` es una constante apuntadora que contiene la dirección de `nums[0]`, el término `&nums[4]` puede reemplazarse por el término equivalente `nums + 4`. Usando cualquiera de estas formas, el programa 12.8 puede volver a escribirse en el programa 12.9 para continuar añadiendo elementos del arreglo mientras la dirección en `nPt` sea menor que o igual a la dirección del último elemento del arreglo.

En el programa 12.9 se usó la forma compacta de la expresión de acumulación, `total += *nPt++`, en lugar de la forma más larga, `total = total + *nPt++`. Además, la expresión `nums + 4` no cambia la dirección en `nums`. Dado que `nums` es un nombre de arreglo y no una variable apuntadora, su valor no puede cambiarse. La expresión `nums + 4` recupera primero la dirección en `nums`, suma 4 a esta dirección (escalada de manera apropiada) y usa el resultado con propósitos de comparación. Expresiones como `*nums++`, la cual intenta cambiar la dirección, son inválidas. Expresiones como `*nums` o `*(nums + i)`, las cuales usan la dirección sin intentar alterarla, son válidas.

Programa 12.9

```
#include <iostream>
using namespace std;

int main()
{
  const int NUMS = 5;

  int nums[NUMS] = {16, 54, 7, 43, -5};
  int total = 0, *nPt;

  nPt = nums;     // almacena la dirección de nums[0] en nPt
  while (nPt < nums + NUMS)
    total += *nPt++;
  cout << "El total de los elementos del arreglo es " << total << endl;

  return 0;
}
```

Inicialización de apuntadores

Como todas las variables, los apuntadores pueden inicializarse cuando se declaran. Sin embargo, cuando se inicializan apuntadores debe tenerse cuidado en establecer una dirección en el apuntador. Por ejemplo, una inicialización como

```
int *ptNum = &millas;
```

sólo es válida si `millas` en sí es declarada como una variable en número entero antes que `ptNum`. Aquí se crea un apuntador a un número entero y se establece la dirección en el apuntador a la dirección de una variable en número entero. Si la variable `millas` se declara después de declarar `ptNum`, como sigue,

```
int *ptNum = &millas;
int millas;
```

ocurre un error. Esto se debe a que se usa la dirección de `millas` antes de definir `millas`. En vista que el área de almacenamiento reservada para `millas` no se ha asignado cuando se declara `ptNum`, la dirección de millas no existe todavía.

Los apuntadores a arreglos también pueden inicializarse dentro de sus instrucciones de declaración. Por ejemplo, si se ha declarado `voltios` como un arreglo de números de precisión doble, puede usarse cualquiera de las siguientes dos declaraciones para inicializar el apuntador nombrado `zing` a la dirección del primer elemento en `voltios`:

```
double *zing = &voltios[0];
double *zing = voltios;
```

La última inicialización es correcta porque `voltios` es en sí misma una constante apuntadora que contiene una dirección del tipo apropiado. (En este ejemplo se seleccionó el nombre de la variable `zing` para reforzar la idea que cualquier nombre de variable puede seleccionarse para un apuntador.)

Ejercicios 12.3

1. Reemplace la instrucción `while` en el programa 12.9 con una instrucción `for`.
2. **a.** Escriba un programa que almacene los siguientes números en el `arreglo` nombrado tasas: 6.25, 6.50, 6.8, 7.2, 7.35, 7.5, 7.65, 7.8, 8.2, 8.4, 8.6, 8.8, 9.0. Despliegue los valores en el arreglo cambiando la dirección en un apuntador llamado `dispPt`. Use una instrucción `for` en su programa.
   b. Modifique el programa escrito en el ejercicio 2a para usar una instrucción `while`.
3. Escriba un programa que almacene los siguientes números en el arreglo llamado `millas`: 15, 22, 16, 18, 27, 23, 20. Haga que su programa copie los datos almacenados en `millas` a otro arreglo llamado `dist` y luego despliegue los valores en el arreglo `dist`. Haga que su programa use una notación de apuntador cuando copie y despliegue elementos del arreglo.
4. Escriba un programa que declare tres arreglos unidimensionales llamados `millas, galones` y `mpg`. Cada arreglo deberá ser capaz de contener diez elementos. En el arreglo `millas` almacene los números 240.5, 300.0, 189.6, 310.6, 280.7, 216.9, 199.4, 160.3, 177.4, 192.3. En el arreglo `galones` almacene los números 10.3, 15.6, 8.7, 14, 16.3, 15.7, 14.9, 10.7, 8.3, 8.4. Cada elemento del arreglo `mpg` deberá calcularse como el elemento correspondiente del arreglo `millas` dividido entre el elemento equivalente del arreglo `galones`: por ejemplo, `mpg[0] = millas[0] / galones[0]`. Use apuntadores cuando calcule y despliegue los elementos del arreglo `mpg`.
5. Defina un arreglo de diez apuntadores a números de precisión doble. Luego lea diez números en las ubicaciones individuales referenciadas por los apuntadores. Ahora sume todos los números y almacene el resultado en una ubicación referenciada a un apuntador. Despliegue el contenido de todas las ubicaciones.

12.4 TRANSMISIÓN DE DIRECCIONES

Ya se ha visto un método para transmitir direcciones a una función. Esto se logró usando variables de referencia, como se describió en la sección 6.3. Aunque la transmisión de variables de referencia a una función le proporciona a ésta la dirección de las variables trans-

mitidas, es un uso implícito de las direcciones debido a que la llamada a la función no revela el hecho que se están usando variables de referencia. Por ejemplo, la llamada a la función `intercambio(num1, num2);` no revela si `num1` o `num2` es una variable de referencia. Sólo al observar la declaración para estas variables o examinar la línea de encabezado de la función para `intercambio()` se revelan los tipos de datos de `num1` y `num2`.

En contraste con la transmisión implícita de direcciones usando variables de referencia, las direcciones pueden transmitirse de manera explícita usando variables apuntadoras. Veamos cómo se logra esto.

Para transmitir de manera explícita una dirección a una función todo lo que se necesita hacer es colocar el operador de dirección, `&`, enfrente de la variable que se está transmitiendo. Por ejemplo, la llamada a la función

```
intercambio(&primernum, &segundonum);
```

transmite las direcciones de las variables `primernum` y `segundonum` a `intercambio()`, como se ilustra en la figura 12.17. Transmitir direcciones en forma explícita usando el operador de dirección en efecto es una *transmisión por referencia* debido a que la función llamada puede hacer referencia, o tener acceso, a las variables en la función que llama usando las direcciones transmitidas. Como se vio en la sección 6.3, las llamadas por referencia también se logran usando parámetros de referencia. Aquí se usarán las direcciones transmitidas y apuntadores para tener acceso directo a las variables `primernum` y `segundonum` desde adentro de `intercambio()` y permutar sus valores, un procedimiento que se logró con anterioridad en el programa 6.8 usando parámetros de referencia.

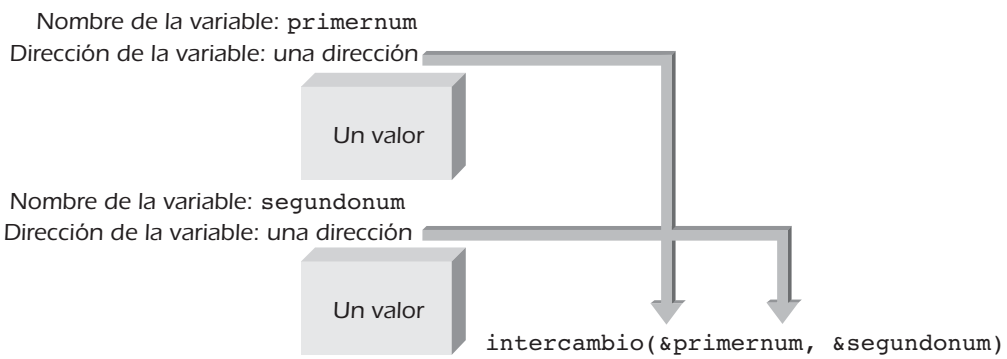


Figura 12.17 *Transmisión explícita de direcciones a* `intercambio()`.

Uno de los primeros requisitos al escribir `intercambio()` es construir una línea de encabezado de función que reciba y almacene en forma correcta los valores transmitidos, que en este caso son dos direcciones. Como se vio en la sección 12.1, las direcciones se almacenan en apuntadores, lo cual significa que los parámetros de `intercambio()` deben declararse como apuntadores.

Suponiendo que `primernum` y `segundonum` son variables de precisión doble, y que `intercambio()` no devuelve un valor, una línea de encabezado de función adecuada para `intercambio` es

```
void intercambio(double *dirNm1, double *dirNm2);
```

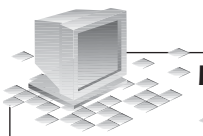
La elección de los nombres de parámetros `dirNm1` y `dirNm2`, como todos los nombres de parámetros, le corresponde al programador. Sin embargo, la declaración `double *dirNm1` declara que el parámetro llamado `dirNm1` se usará para almacenar la dirección

de un valor de precisión doble. Del mismo modo, la declaración `double *dirNm2` declara que `dirNm2` también almacenará la dirección de un valor de precisión doble.

Antes de escribir el cuerpo de `intercambio()` para permutar los valores en `primernum` y `segundonum`, se comprobará primero que los valores a los que se tiene acceso usando las direcciones en `dirNm1` y `dirNm2` son correctos. Esto se hace en el programa 12.10.

La salida desplegada cuando se ejecuta el programa 12.10 es

```
El número cuya dirección está en dirNm1 es 20.5
El número cuya dirección está en dirNm2 es 6.25
```



Programa 12.10

```
#include <iostream>
using namespace std;

void intercambio(double *, double *);      // prototipo de la función

int main()
{
  double primernum = 20.5, segundonum = 6.25;

  intercambio(&primernum, &segundonum);    // llama a intercambio

  return 0;
}

// esta función ilustra la transmisión de argumentos apuntadores
void intercambio(double *dirNm1, double *dirNm2)
{

  cout << "El número cuya dirección está en dirNm1 es "
       << *dirNm1 << endl;
  cout << "El número cuya dirección está en dirNm1 es "
       << *dirNm2 << endl;

  return;
}
```

Al revisar el programa 12.10 se notarán dos cosas. Primera, el prototipo de la función para `intercambio()`

```
void intercambio(double * , double*)
```

declara que `intercambio()` no devuelve ningún valor en forma directa y que sus parámetros son dos apuntadores que "apuntan" a valores de precisión doble. Como tal, cuando se llame a la función ésta requerirá que se transmitan dos direcciones, y que cada una sea la dirección de un valor de precisión doble.

Segunda, que dentro de `intercambio()` se usa el operador de indirección para tener acceso a los valores almacenados en `primernum` y `segundonum`. La función `inter-`

`cambio()` en sí no tiene conocimiento de estos nombres de variables, pero tiene la dirección de `primernum` almacenado en `dirNm1` y la dirección de `segundonum` almacenada en `dirNm2`. La expresión `*dirNm1` usada en la primera instrucción `cout` significa "la variable cuya dirección está en `dirNm1`". Ésta es por supuesto la variable `primernum`. Del mismo modo, la segunda instrucción `cout` obtiene el valor almacenado en `segundonum` como "la variable cuya dirección está en `dirNm2`". Por tanto, se han usado con éxito apuntadores para permitir que `intercambio()` tenga acceso a variables en `main()`. La figura 12.18 ilustra el concepto de almacenar direcciones en parámetros.

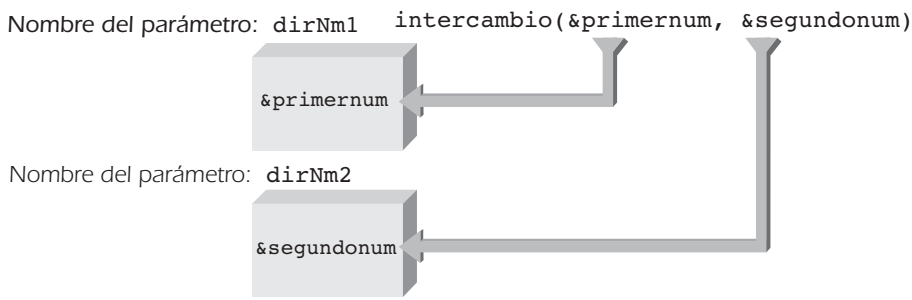


Figura 12.18 *Almacenar direcciones en parámetros.*

Habiendo verificado que `intercambio()` puede tener acceso a las variables locales de `main()` `primernum` y `segundonum`, ahora se puede expandir `intercambio()` para permutar los valores en estas variables. Los valores en las variables `primernum` y `segundonum` de `main()` pueden intercambiarse desde dentro de `intercambio()` usando el algoritmo de intercambio de tres pasos descrito antes en la sección 6.3, el cual por comodidad se muestra de nuevo a continuación:

1. Almacenar el valor de `primernum` en una ubicación temporal.
2. Almacenar el valor de `segundonum` en `primernum`.
3. Almacenar el valor temporal en `segundonum`.

Usando apuntadores desde dentro de `intercambio()`, esto adopta la forma:

1. Almacenar el valor de la variable a la que apunta `dirNm1` en una ubicación temporal. La instrucción `temp = *dirNm1;` hace esto (véase la figura 12.19).
2. Almacenar el valor de la variable cuya dirección está en `dirNm2` en la variable cuya dirección están en `dirNm1`. La instrucción `*dirNm1 = *dirNm2;` hace esto (véase la figura 12.20).
3. Mover el valor en la ubicación temporal a la variable cuya dirección está en `dirNm2`. La instrucción `*dirNm2 = temp;` hace esto (véase la figura 12.21).

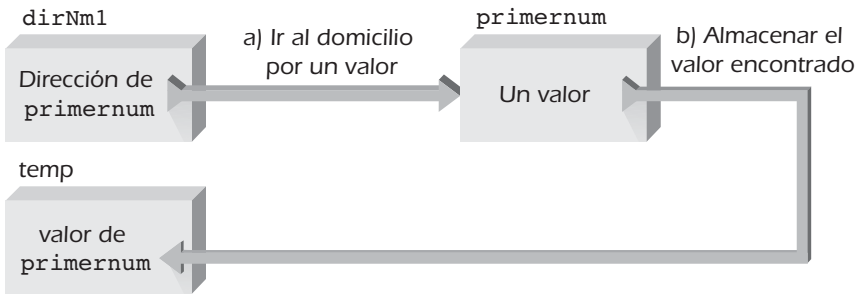


Figura 12.19 *Almacenar el valor de* `primernum` *en forma indirecta.*

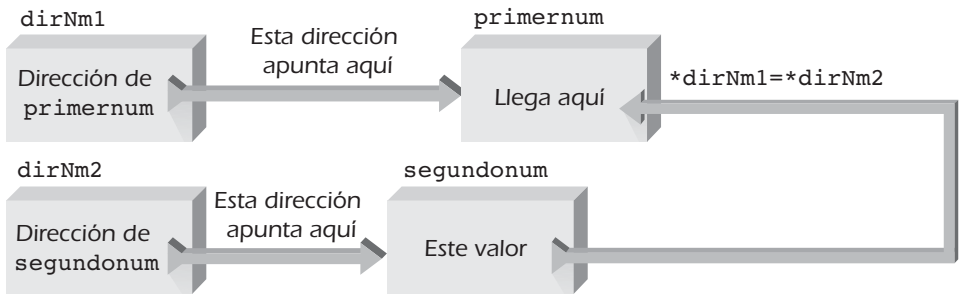


Figura 12.20 *Cambiar de manera indirecta el valor de* `primernum`.

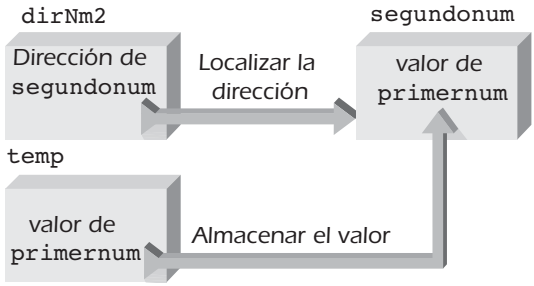


Figura 12.21 *Cambiar en forma indirecta el valor de* `segundonum`.

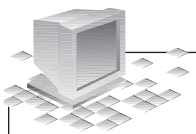
El programa 12.11 contiene la forma final de `intercambio()`, escrito de acuerdo con nuestra descripción.

La siguiente muestra de ejecución se obtuvo usando el programa 12.11:

```
El valor almacenado en primernum es: 20.5
El valor almacenado en segundonum es: 6.25

El valor almacenado ahora en primernum es: 6.25
El valor almacenado ahora en segundonum es: 20.5
```

Como lo ilustra esta salida, los valores almacenados en las variables de `main()` se han modificado desde dentro de `intercambio()`, lo cual se hizo posible por el uso de apuntadores. El lector interesado deberá comparar esta versión de `intercambio()` con la versión que utiliza referencias que se presentó en el programa 6.10. La ventaja de usar apuntadores en lugar de referencias es que la llamada a la función en sí designa de manera explícita qué direcciones se van a usar, lo cual es una alerta directa de que la función con toda probabilidad alterará variables de la función que llama. Las ventajas de usar referencias es que la notación es mucho más simple.

En general, para funciones como inter`cambio()`, gana la conveniencia de la notación y se usan referencias. Sin embargo, al transmitir arreglos a funciones, lo cual es el siguiente tema, el compilador transmite de manera automática una dirección. Esto dicta que se usarán variables apuntadoras para almacenar la dirección.



Programa 12.11

```
#include <iostream>
using namespace std;

void intercambio(double *, double *);      // prototipo de la función

int main()
{
  double primernum = 20.5, segundonum = 6.25;


  cout << "El valor almacenado en primernum es: " << primernum << endl;
  cout << "El valor almacenado en segundonum es: " << segundonum << "\n\n";

  intercambio(&primernum, &segundonum);    // llama a intercambio

  cout << "El valor almacenado ahora en primernum es: "
       << primernum << endl;
  cout << "El valor almacenado ahora en segundonum es: "
       << segundonum << endl;

  return 0;
}

// esta función intercambia los valores en sus dos argumentos
void intercambio(double *dirNm1, double *dirNm2)
{
  double temp;
```

(Continúa)

(Continuación)

```
   temp = *dirNm1;              // guarda el valor de primernum
   *dirNm1 = *dirNm2;           // mueve el valor de segundonum a primernum
   *dirNm2 = temp;              // cambia el valor de segundonum

   return;
}
```

Transmisión de arreglos

Cuando se transmite un arreglo a una función, su dirección es el único elemento que se transmite en realidad. Con esto se quiere decir la dirección de la primera ubicación usada para almacenar el arreglo, como se ilustra en la figura 12.22. Dado que la primera ubicación reservada para un arreglo corresponde al elemento 0 del arreglo, la "dirección del arreglo" también es la dirección del elemento 0.

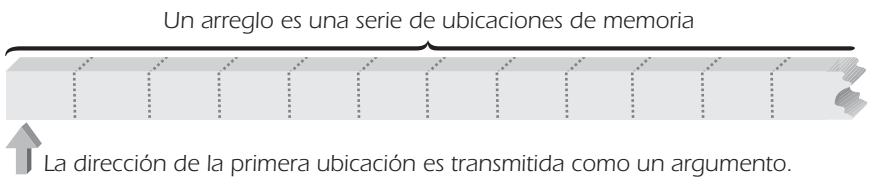
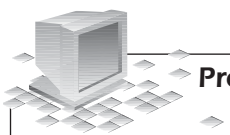


Figura 12.22 *La dirección de un arreglo es la dirección de la primera ubicación reservada para el arreglo.*

Para proporcionar un ejemplo específico en el cual un arreglo es transmitido a una función, considérese el programa 12.12. En este programa, el arreglo `nums` es transmitido a la función `hallarMax()` usando notación de arreglos convencional.



Programa 12.12

```
#include <iostream>
using namespace std;

int hallarMax(int [], int); // prototipo de la función

int main()
{
  const int NUMPTS = 5;

  int nums[NUMPTS] = {2, 18, 1, 27, 16};
```

(Continúa)

(Continuación)

```
  cout << "\nEl valor máximo es "
       << hallarMax(nums,NUMPTS) << endl;
  return 0;
}
// esta función devuelve el valor máximo en un arreglo de números enteros
int hallarMax(int vals[], int numels)
{
  int i, max = vals[0];

  for (i = 1; i < numels; i++)
   if (max < vals[i])
     max = vals[i];

  return max;
}
```

La salida desplegada cuando se ejecuta el programa 12.12 es

```
El valor máximo es 27
```

El parámetro llamado `vals` en la declaración de la línea de encabezado para `hallarMax()` en realidad recibe la dirección del arreglo `nums`. Como tal, `vals` en realidad es un apuntador, dado que los apuntadores son variables (o parámetros) usados para almacenar direcciones. En vista que la dirección transmitida a `hallarMax()` es la dirección de un número entero, otra línea de encabezado adecuada para `hallarMax()` es

```
int hallarMax(int *vals, int numels) // aquí vals se declara como
                                     // un apuntador a un numero entero
```

La declaración `int *vals` en la línea de encabezado declara que `vals` se usa para almacenar una dirección de un número entero. La dirección almacenada es, por supuesto, la ubicación del inicio de un arreglo. La siguiente es una versión replanteada de la función `hallarMax()` que usa la nueva declaración de apuntador para `vals`, pero conserva el uso de subíndices para referirse a elementos individuales del arreglo:

```
int hallarMax(int *vals, int numels) // encuentra el valor máximo
{
  int i, max = vals[0];

  for (i = 1; i < numels; i++)
   if (max < vals[i])
     max = vals[i];

  return max;
}
```

Sin tomar en cuenta cómo se declaró `vals` en el encabezado de la función o cómo se usa dentro del cuerpo de la función, en verdad es una variable apuntadora. Por tanto, la dirección en `vals` puede modificarse. Esto no sucede con el nombre `nums`. Dado que nums es el nombre del arreglo originalmente creado, es una constante apuntadora. Como se describió en la sección 12.2, esto significa que la dirección en `nums` no puede cambiarse y que

no puede tomarse la dirección de `nums` en sí. Sin embargo, ninguna de estas restricciones se aplican a la variable apuntadora llamada `vals`. Toda la aritmética de direcciones que se aprendió en la sección anterior puede aplicarse de manera legítima a `vals`.

Escribiremos dos versiones adicionales de `hallarMax()`, ambas usando apuntadores en lugar de subíndices. En la primera versión tan sólo se sustituye la notación de apuntador por notación de subíndice. En la segunda versión usaremos aritmética de direcciones para cambiar la dirección en el apuntador.

Como ya se señaló, el acceso a un elemento del arreglo usando la notación de subíndice `nombreArreglo[i]` siempre puede reemplazarse por la notación de apuntador `*(nombreArreglo + i)`. En la primera modificación a `hallarMax()`, se puede usar esta correspondencia con sólo reemplazar todas las referencias a `vals[i]` con la expresión equivalente `*(vals + i)`.

```
int hallarMax(int *vals, int numels) // halla el valor máximo
{
  int i, max = *vals;

  for (i = 1; i < numels; i++)
   if (max < *(vals + i) )
     max = *(vals + i);

  return max;
}
```

La siguiente versión de `hallarMax()` aprovecha el hecho que puede cambiarse la dirección almacenada en `vals`. Después que se recupera cada elemento del arreglo usando la dirección en `vals`, la dirección en sí se incrementa en uno en la lista que se altera en la instrucción `for`. La expresión `max = *vals` usada antes para establecer `max` con el valor de `vals[0]` es reemplazada por la expresión `max = *vals++`, la cual ajusta la dirección en vals para que apunte al segundo elemento en el arreglo. El elemento asignado a `max` por esta expresión es el elemento del arreglo al que apunta `vals` antes que se incremente `vals`. El posfijo de incremento, `++`, no cambia la dirección en `vals` hasta después que se ha usado la dirección para recuperar el primer elemento del arreglo.

```
int hallarMax(int *vals, int numels) // encuentra el valor máximo
{
  int i, max = *vals++; // obtiene el primer elemento y lo
                        incrementa
  for (i = 1; i < numels; i++, vals++)
  {
    if (max < *vals)
      max = *vals;
  }
  return max;
}
```

Ahora se revisará esta versión de `hallarMax()`. Al principio el valor máximo se establece como "el elemento al que apunta `vals`". Dado que `vals` contiene al principio la dirección del primer elemento en el arreglo transmitido a `hallarMax()`, el valor de este primer elemento se almacena en `max`. La dirección en `vals` se incrementa entonces en uno. El uno que se suma a `vals` se escala de manera automática con el número de bytes

usados para almacenar números enteros. Por tanto, después del incremento, la dirección almacenada en `vals` es la dirección del siguiente elemento del arreglo. Esto se ilustra en la figura 12.23. El valor de este siguiente elemento se compara con el máximo y la dirección se incrementa de nuevo, esta vez desde dentro de la lista de alteración de la instrucción `for`. Este proceso continúa hasta que se han examinado todos los elementos del arreglo.

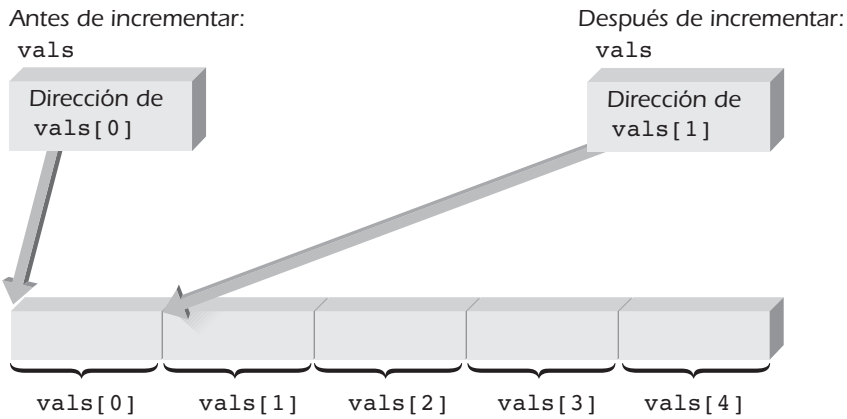


Figura 12.23 *Apuntando a diferentes elementos.*

La versión de `hallarMax()` que deberá elegir es cuestión de estilo y gusto personales. En general, los programadores principiantes se sienten más cómodos usando subíndices en lugar de apuntadores. Además, si el programa usa un arreglo como la estructura de almacenamiento natural para la aplicación y los datos disponibles, el acceso a un arreglo usando subíndices es más apropiado para indicar con claridad la intención del programa. Sin embargo, conforme se aprende sobre estructuras de datos, el uso de apuntadores se vuelve una herramienta cada vez más útil y poderosa por su propio derecho. En estos casos no hay una equivalencia simple o fácil para el uso de subíndices.

Puede deducirse un "truco ingenioso" de esta exposición. Dado que la transmisión de un arreglo a una función en realidad implica la transmisión de una dirección, se puede transmitir igual de bien cualquier dirección válida. Por ejemplo, la llamada a la función `hallarMax(&nums[2],3)` transmite la dirección de `nums[2]` a `hallarMax()`. Dentro de `hallarMax()` el apuntador `vals` almacena la dirección y la función comienza la búsqueda de un máximo en el elemento correspondiente a esta dirección. Por tanto, desde la perspectiva de `hallarMax()`, ha recibido una dirección y procede en forma apropiada.

Notación avanzada para apuntadores[9]

También puede tenerse acceso a arreglos multidimensionales usando notación de apuntador, aunque la notación se vuelve cada vez más críptica conforme aumentan las dimensiones del

[9]Este tema puede omitirse sin perder la continuidad temática.

arreglo. Una aplicación muy útil de esta notación ocurre con los arreglos bidimensionales de caracteres. Aquí se considerará la notación de apuntador para arreglos bidimensionales numéricos. Por ejemplo, considérese la declaración

```
int nums[2][3] = { {16,18,20},
                   {25,26,27} };
```

Esta declaración crea un arreglo de elementos y un conjunto de constantes apuntadoras llamadas nums, `nums[0]` y `nums[1]`. La relación entre estas constantes apuntadoras y los elementos del arreglo `nums` se ilustra en la figura 12.24.

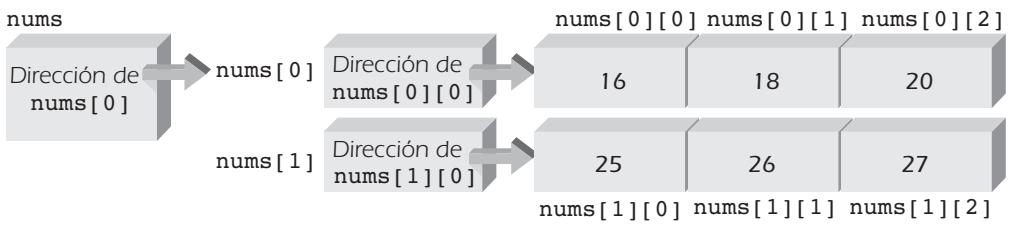


Figura 12.24 *Almacenamiento del arreglo* `nums` *y constantes apuntadoras asociadas.*

La disponibilidad de las constantes apuntadoras asociadas con un arreglo bidimensional permite tener acceso a los elementos del arreglo en una variedad de formas. Una de ellas es considerar el arreglo bidimensional como un arreglo de filas, donde cada fila en sí misma es un arreglo de tres elementos. Considerado bajo esta luz, la dirección del primer elemento en la primera fila es proporcionada por `nums[0]` y la dirección del primer elemento en la segunda fila es proporcionada por `nums[1]`. Por tanto, la variable a la que apunta `nums[0]` es `nums[0][0]` y la variable a la que apunta `nums[1]` es `nums[1][0]`. Una vez que se entiende la naturaleza de estas constantes, puede tenerse acceso a cada elemento en el arreglo al aplicar un desplazamiento apropiado al apuntador apropiado. Por tanto, las siguientes notaciones son equivalentes:

Notación de apuntador	Notación de subíndice	Valor
`*nums[0]`	`nums[0][0]`	16
`*(nums[0] + 1)`	`nums[0][1]`	18
`*(nums[0] + 2)`	`nums[0][2]`	20
`*nums[1]`	`nums[1][0]`	25
`*(nums[1] + 1)`	`nums[1][1]`	26
`*(nums[1] + 2)`	`nums[1][2]`	27

Ahora es posible avanzar aún más y reemplazar `nums[0]` y `nums[1]` con sus respectivas notaciones de apuntador, usando la dirección de la misma `nums`. Como se ilustra en la figura 12.24, la variable a la que apunta nums es `nums[0]`. Es decir, `*nums es nums[0]`.

Del mismo modo, `*(nums + 1)` es `nums[1]`. Usar estas relaciones conduce a las siguientes equivalencias:

Notación de apuntador	Notación de subíndice	Valor
`*(*nums)`	`nums[0][0]`	16
`*(*nums + 1)`	`nums[0][1]`	18
`*(*nums + 2)`	`nums[0][2]`	20
`*(*(nums + 1))`	`nums[1][0]`	25
`*(*(nums + 1) + 1)`	`nums[1][1]`	26
`*(*(nums + 1) + 2)`	`nums[1][2]`	27

Se aplica la misma notación cuando se transmite un arreglo bidimensional a una función. Por ejemplo, supóngase que el arreglo bidimensional `nums` se transmite a la función `calc()` usando la llamada `calc(nums);`. Aquí, como con todas las transmisiones de arreglos, se transmite una dirección. Una línea de encabezado de la función adecuada para la función `calc()` es

```
calc(int pt[2][3])
```

Como ya se ha visto, la declaración de parámetros para `pt` también puede ser

```
calc(int pt[][3])
```

Usando notación de apuntador, otra declaración adecuada es

```
calc(int (*pt)[3])
```

En esta última declaración se requieren los paréntesis interiores para crear un apuntador único para arreglos de tres números enteros. Por supuesto, cada arreglo es equivalente a una sola fila del arreglo `nums`. Al desplazar de manera adecuada el apuntador, puede tenerse acceso a cada elemento en el arreglo. Hay que observar que sin los paréntesis la declaración se vuelve

```
int *pt[3]
```

lo cual crea un arreglo de tres apuntadores, cada uno apuntando a un solo número entero.

Una vez que se hace la declaración correcta para `pt` (puede usarse cualquiera de las tres declaraciones válidas), todas las notaciones siguientes dentro de la función `calc()` son equivalentes:

Notación de apuntador	Notación de subíndice	Valor
`*(*pt)`	`pt[0][0]`	16
`*(*pt+1)`	`pt[0][1]`	18
`*(*pt+2)`	`pt[0][2]`	20
`*(*(pt+1))`	`pt[1][0]`	25
`*(*(pt+1)+1)`	`pt[1][1]`	26
`*(*(pt+1)+2)`	`pt[1][2]`	27

Las últimas dos notaciones usando apuntadores se encuentran en programas en C++ más avanzados. La primera de éstas ocurre debido a que las funciones pueden devolver cualquier tipo de datos escalares en C++ válido, incluyendo apuntadores a cualquiera de estos tipos de datos. Si una función devuelve un apuntador, el tipo de datos al que se está apuntando debe declararse en la declaración de la función. Por ejemplo, la declaración

```
int *calc()
```

declara que `calc()` devuelve un apuntador a un valor en número entero. Esto significa que se devuelve una dirección de una variable en número entero. Del mismo modo, la declaración

```
double *impuestos()
```

declara que `impuestos()` devuelve un apuntador a un valor de precisión doble. Esto significa que se devuelve una dirección de una variable de precisión doble.

Además de declarar apuntadores a números enteros, números de precisión doble y otros tipos de datos de C++, también pueden declararse apuntadores que apunten a (que contengan la dirección de) una función. Los apuntadores a funciones son posibles debido a que los nombres de función, como los nombres de arreglos, son en sí mismos constantes apuntadoras. Por ejemplo, la declaración

```
int (*calc)()
```

declara que `calc()` es un apuntador a una función que devuelve un número entero. Esto significa que `calc` contendrá la dirección de una función, y la función cuya dirección está en la variable `calc` devuelve un valor en número entero. Por ejemplo, si la función `suma()` devuelve un número entero, es válida la asignación `calc = suma;`.

Ejercicios 12.4

1. La siguiente declaración se usó para crear el arreglo `precios`:

   ```
   double precios[500];
   ```

   Escriba tres encabezados diferentes para una función llamada `ordenarArreglo()` que acepte el arreglo precios como un parámetro llamado `enArreglo` y no devuelva ningún valor.

2. La siguiente declaración se usó para crear el arreglo `claves`:

   ```
   char claves[256];
   ```

   Escriba tres encabezados diferentes para una función llamada `hallarClave()` que acepte el arreglo `claves` como un parámetro llamado seleccionar y no devuelva ningún valor.

3. La siguiente declaración se usó para crear el `arreglo` tasas:

   ```
   double tasas[256];
   ```

   Escriba tres encabezados diferentes para una función llamada `maximo()` que acepte el arreglo `tasas` como un parámetro llamado `velocidad` y devuelva un valor de precisión doble.

4. Modifique la función `hallarMax()` para localizar el valor mínimo del arreglo transmitido. Escriba la función usando sólo apuntadores.

5. En la última versión de `hallarMax()` presentada, `vals` se incrementaba dentro de la lista de alteración de la instrucción `for`. En cambio, suponga que se hace el incremento dentro de la expresión condicional de la instrucción `if` como sigue:

```
int hallarMax(int *vals, int numels)    // versión incorrecta
{
  int i, max = *vals++; // obtiene el primer elemento y se
                           incrementa

  for (i = 1; i < numels; i++)
    if (max < *vals++)
      max = *vals;
  return (max);
}
```

Esta versión produce un resultado incorrecto. Determine por qué.

6. a. Escriba un programa que tenga una declaración en `main` para almacenar los siguientes números en un arreglo llamado `tasas`: 6.5, 7.2, 7.5, 8.3, 8.6, 9.4, 9.6, 9.8, 10.0. Deberá haber una llamada a la función `mostrar()` que acepte tasas en un parámetro llamado `tasas` y luego despliegue los números usando la notación de apuntador `*(tasas + i)`.

b. Modifique la función `mostrar()` escrita en el ejercicio 6a para alterar la dirección en `tasas`. Use siempre la expresión `*tasas` en lugar de `*(tasas + i)` para recuperar el elemento correcto.

7. a. Determine la salida del siguiente programa:

```
#include <iostream>
using namespace std;

const int FILAS = 2;
const int COLS = 3;

void arr(int [][COLS]);

int main()
{
  int nums[FILAS][COLS] = { {33, 16, 29},
                            {54,67,99}};
  arr(nums);

  return 0;
}
```

```
void arr(int (*val)[COLS])
{
  cout << '\n' << *(*val);
  cout << '\n' << *(*val + 1);
  cout << '\n' << *(*(val + 1) + 2);
  cout << '\n' << *(*val) + 1;

  return;
}
```

b. Dada la declaración para `val` en la función `arr()`, ¿sería válida la notación `val[1][2]` dentro de la función?

12.5 Errores comunes de programación

Al usar el material presentado en este capítulo, debe percatarse de los siguientes posibles errores.

1. Intentar almacenar una dirección en una variable que no ha sido declarada como un apuntador.
2. Usar un apuntador para tener acceso a elementos inexistentes de un arreglo. Por ejemplo, si `nums` es un arreglo de diez números enteros, la expresión `*(nums + 15)` apunta a una ubicación que está seis ubicaciones de número entero adelante del último elemento del arreglo. Debido a que C++ no hace ninguna comprobación de límites en accesos a arreglos, este tipo de error no es detectado por el compilador. Éste es el mismo error, disfrazado en forma de notación de apuntador, que ocurre cuando se usa un subíndice para tener acceso a un elemento del arreglo fuera de los límites.
3. Olvidar usar el conjunto de corchetes, `[]`, después del operador `delete` cuando se desasigna en forma dinámica la memoria que fue asignada antes usando el nuevo operador `[]`.
4. Aplicar de manera incorrecta los operadores de dirección e indirección. Por ejemplo, si `pt` es una variable apuntadora, las expresiones

   ```
   pt = &45
   pt = &(millas + 10)
   ```

   son inválidas porque intentan tomar la dirección de un valor. Sin embargo, hay que observar que la expresión `pt = &millas + 10` es válida. Aquí, 10 se suma a la dirección de `millas`. Una vez más, es responsabilidad del programador asegurar que la dirección final "apunte a" un elemento de datos válido.
5. Tomar direcciones de constantes apuntadoras. Por ejemplo, dadas las declaraciones

   ```
   int nums[25];
   int *pt;
   ```

la asignación

```
pt = &nums;
```

es inválida. La constante `nums` es una constante apuntadora que en sí misma es equivalente a una dirección. La asignación correcta es `pt = nums`.

6. Tomar direcciones de un argumento de referencia, variable de referencia o variable de registro. La razón para esto es que los argumentos y variables de referencia son en esencia lo mismo que constantes apuntadoras, en cuanto a que son valores de dirección nombrados. Del mismo modo, no puede tomarse la dirección de una variable de registro. Por tanto, para las declaraciones

   ```
   register in total;
   int *ptTot;
   ```

   la asignación

   ```
   ptTot = &total;  // INVALIDO
   ```

   es inválida. La razón es que las variables de registro se almacenan en los registros internos de la computadora, y estas áreas de almacenamiento no tienen direcciones de memoria estándar.

7. Inicializar variables apuntadoras en forma incorrecta. Por ejemplo, la inicialización

   ```
   int *pt = 5;
   ```

   es inválida. Dado que `pt` es un apuntador a un número entero, debe inicializarse con una dirección válida.

8. Confundirse respecto a si una variable *contiene* una dirección o es una dirección. Las variables apuntadoras y los argumentos apuntadores contienen direcciones. Aunque una constante apuntadora es sinónimo de una dirección, es útil tratar a las constantes apuntadoras como variables apuntadoras con dos restricciones:
   - La dirección de una constante apuntadora no puede ser toamda.
   - La dirección "contenida en" la constante apuntadora no puede ser alterada.

   Excepto por estas dos restricciones, las constantes apuntadoras y las variables apuntadoras pueden usarse en forma casi intercambiable. Por consiguiente, cuando se requiere una dirección, puede usarse cualquiera de los siguientes:
   - un nombre de variable apuntadora
   - un nombre de argumento apuntador
   - un nombre de constante apuntadora
   - un nombre de variable no apuntadora precedido por el operador de dirección (por ejemplo, `&variable`)
   - un nombre de argumento no apuntador precedido por el operador de dirección (por ejemplo, `&argumento`)

   Algo de la confusión que rodea a los apuntadores es causada por el uso indiferente de la palabra *apuntador*. Por ejemplo, el enunciado "una función requiere un argumento apuntador" se entiende con más claridad cuando uno se da cuenta que el enunciado significa en realidad "una función requiere una dirección como argumento". Del mismo modo, el enunciado "una función devuelve un apuntador" en realidad significa "una función devuelve una dirección".

Si en alguna ocasión tiene dudas respecto a lo que contiene en realidad una variable o cómo deberá ser tratada, use el objeto `cout` para desplegar el contenido de la variable, el "elemento al que apunta" o "la dirección de la variable". Poder observar lo que se despliega con frecuencia ayuda a aclarar lo que hay en realidad en la variable.

12.6 RESUMEN DEL CAPÍTULO

1. Toda variable tiene un tipo de datos, una dirección y un valor. En C++ puede obtenerse la dirección de una variable usando el operador de dirección `&`.
2. Un apuntador es una variable que se utiliza para almacenar la dirección de otra variable. Los apuntadores, como todas las variables en C++, deben declararse. El operador de indirección, `*`, se usa tanto para declarar una variable apuntadora como para tener acceso a la variable cuya dirección se almacena en un apuntador.
3. Un nombre de arreglo es una constante apuntadora. El valor de la constante apuntadora es la dirección del primer elemento en el arreglo. Por tanto, si `val` es el nombre de un arreglo, `val` y `&val[0]` pueden usarse de manera intercambiable.
4. Cualquier acceso a un elemento del arreglo usando notación de subíndice siempre puede reemplazarse usando notación de apuntador. Es decir, la notación `a[i]` siempre puede reemplazarse con la notación `*(a + i)`. Esto es cierto aun si `a` fue declarada inicialmente en forma explícita como un arreglo o como un apuntador.
5. Los arreglos pueden crearse en forma dinámica mientras se está ejecutando un programa. Por ejemplo, la secuencia de instrucciones

   ```
   cout << "Introduzca el tamanio del arreglo: ";
   cin   >> num;
   int +calif = new int[num];
   ```

   crea un nombre de arreglo `grades` del tamaño `num`. El área asignada para el nombre de arreglo puede ser destruido dinámicamente usando el operador `delete[]`. Por ejemplo, la declaración `delete[] grades;` devolvera el área asignada para el nombre de arreglo de `grades` de nuevo a la computadora.
6. Los arreglos se transmiten a las funciones como direcciones. La función llamada siempre recibe acceso directo a los elementos del arreglo declarado originalmente.
7. Cuando se transmite un arreglo unidimensional a una función, la declaración de parámetros para la función puede ser una declaración de arreglo o una declaración de apuntador. Por tanto, las siguientes declaraciones de parámetros son equivalentes:

   ```
   double a[];
   double *a;
   ```

8. Los apuntadores pueden incrementarse, disminuirse, compararse y asignarse. Los números sumados o restados de un apuntador se escalan de manera automática. El factor de escala usado es el número de bytes requeridos para almacenar el tipo de datos al que se apunta originalmente.

CAPÍTULO 13
Estructuras

TEMAS

13.1 ESTRUCTURAS SENCILLAS

13.2 ARREGLOS DE ESTRUCTURAS

13.3 ESTRUCTURAS COMO ARGUMENTOS DE FUNCIÓN
TRANSMISIÓN DE UN APUNTADOR
DEVOLUCIÓN DE ESTRUCTURAS

13.4 LISTAS VINCULADAS

13.5 ASIGNACIÓN DINÁMICA DE ESTRUCTURAS DE DATOS

13.6 UNIONES

13.7 ERRORES COMUNES DE PROGRAMACIÓN

13.8 RESUMEN DEL CAPÍTULO

Una estructura es un vestigio histórico de C. Desde la perspectiva del programador, una estructura puede considerarse como una clase que tiene todas las variables de instancia pública y no tiene métodos miembros. En las aplicaciones comerciales a la estructura se le denomina como, y es lo mismo que un registro. En C y C++, una estructura proporciona un medio para almacenar valores que tienen diferentes tipos de datos, como un número de parte entero, un tipo parte en carácter y un suministro de voltaje en número de precisión doble.

Por ejemplo, supóngase que un fabricante de circuitos integrados (CI) mantiene un resumen de información para cada uno de los circuitos que fabrica. Los elementos de datos conservados para cada circuito se ilustran en la figura 13.1.

Número de parte:
Familia de circuitos integrados:
Tipo de función:
Suministro de voltaje:
Unidades en existencia:

Figura 13.1 Un registro de inventario.

Cada uno de los elementos de datos individuales enlistados en la figura 13.1 es una entidad en sí mismo que se conoce como un ***campo de datos****. En conjunto, todos los campos de datos forman una sola unidad que se denomina* ***estructura****.*

Aunque el fabricante de circuitos integrados podría dar seguimiento a cientos de componentes, la forma de cada estructura de carácter es idéntica. Al tratar con estructuras es importante distinguir entre la forma de una estructura y su contenido.

La forma de una estructura consiste en los nombres simbólicos, tipos de datos y el orden de los campos de datos individuales en la estructura. El contenido de la estructura se refiere a los datos reales almacenados en los nombres simbólicos. La figura 13.2 muestra contenidos aceptables para la forma de estructura ilustrada en la figura 13.1.

Número de parte: 23421
Familia de circuitos integrados: TTL
Tipo de función: AND
Suministro de voltaje: 6.0
Unidades en existencia: 345

Figura 13.2 La forma y contenido de un registro.

En este capítulo se describen las instrucciones de C++ que se requieren para crear, llenar, usar y transmitir estructuras entre funciones.

13.1 ESTRUCTURAS SENCILLAS

Crear y usar una estructura requiere los mismos dos pasos necesarios para crear y usar cualquier variable. Primero debe declararse la estructura del registro. Luego pueden asignarse valores específicos a los elementos individuales de la estructura. Declarar una estructura requiere enlistar los tipos de datos, los nombres de datos y el orden de los elementos de datos. Por ejemplo, la definición

```
struct
{
  int mes;
  int dia;
  int anio;
} nacimiento;
```

da la forma de una estructura llamada `nacimiento` y reserva almacenamiento para los elementos de datos individuales enlistados en la estructura. La estructura `nacimiento` consiste en tres elementos de datos o campos, los cuales se denominan miembros de la estructura.

Asignar valores de datos reales a los elementos de datos de una estructura se conoce como *poblar la estructura* y es un procedimiento relativamente sencillo. Se tienen acceso a cada miembro de una estructura dando tanto el nombre de la estructura como el nombre del elemento de datos individual, separado por un punto. Por tanto, `nacimiento-.mes` se refiere al primer miembro de la estructura `nacimiento, nacimiento.dia` se refiere al segundo miembro de la estructura y `nacimiento.anio` se refiere al tercer miembro. El programa 13.1 ilustra la asignación de valores a los miembros individuales de la estructura `nacimiento`.

Program 13.1

```
// a program that defines and populates a record
#include <iostream>
using namespace std;

int main()
{
  struct
  {
    int month;
    int day;
    int year;
  } birth;

  birth.month = 12;
  birth.day = 28;
  birth.year = 86;

  cout << "My birth date is "
       << birth.month << '/'
       << birth.day   << '/'
       << birth.year  << endl;
```

La salida producida por el programa 13.1 es

```
Mi fecha de nacimiento es 12/28/86
```

Como en la mayor parte de las instrucciones en C++, el espaciamiento en la definición de una estructura no es rígido. Por ejemplo, la estructura nacimiento podría haberse definido también como

```
struct {int mes; int dia; int anio;} nacimiento;
```

Además, como con todas las definiciones de instrucciones en C++, pueden definirse múltiples variables en la misma instrucción. Por ejemplo, la instrucción de definición

```
struct
{
  int mes;
  int dia;
  int anio;
} nacimiento, actual;
```

crea dos variables de estructura que tienen la misma forma. Se hace referencia a los miembros de la primera estructura con los nombres individuales `nacimiento.mes`, `nacimiento.dia` y `nacimiento.anio`, mientras que se hace referencia a los miembros de la segunda estructura con los nombres `actual.mes, actual.dia` y `actual.anio`. Hay

que observar que la forma de esta instrucción de definición de una estructura particular es idéntica a la forma usada al definir cualquier variable de programa: el tipo de datos es seguido por una lista de nombres de variables.

Una modificación útil y muy usada para definir tipos de estructura es enlistar la forma de la estructura sin que le sigan los nombres de variables. En este caso, sin embargo, la lista de los miembros de la estructura debe ser precedida por un nombre de tipo de datos seleccionado por el usuario. Por ejemplo, en la declaración

```
struct Fecha
{
  int mes;
  int dia;
  int anio;
};
```

el término `Fecha` es un nombre de tipo de estructura: define un tipo de datos nuevo que es una estructura de datos de la forma declarada.[1] Por convención la primera letra de un nombre de tipo de datos seleccionado por el usuario se pone en mayúscula, como en el nombre `Fecha`, lo cual ayuda a identificarla cuando se usa en instrucciones de declaración subsiguientes. Aquí, la declaración para la estructura `Fecha` crea un tipo de datos nuevo sin reservar en realidad ninguna ubicación de almacenamiento. Como tal no es una instrucción de definición. Tan sólo declara un tipo de estructura `Fecha` y describe cómo se ordenan los elementos de datos individuales dentro de la estructura. El almacenamiento real para los miembros de la estructura sólo se reserva cuando se asignan nombres de variable específicos. Por ejemplo, la instrucción de definición

```
Fecha nacimiento, actual;
```

reserva almacenamiento para dos variables de la estructura `Fecha` llamadas `nacimiento` y `actual`, respectivamente. Cada una de estas estructuras individuales tiene la forma declarada con anterioridad para la estructura `Fecha`.

La declaración de los tipos de datos de una estructura, como todas las declaraciones, puede ser global o local. El programa 13.2 ilustra la declaración global de un tipo de datos de `Fecha`. Dentro de `main()`, la variable `nacimiento` se define como una variable local del tipo `Fecha`.

La salida producida por el programa 13.2 es idéntica a la salida producida por el programa 13.1.

La inicialización de estructuras sigue las mismas reglas que para la inicialización de arreglos: las estructuras globales y locales pueden inicializarse al colocar después de la definición una lista de inicializadores. Por ejemplo, la instrucción de definición

```
Fecha nacimiento = {12, 28, 86};
```

puede usarse para reemplazar las primeras cuatro instrucciones internas en `main()` en el programa 13.2. Hay que observar que los inicializadores están separados por comas, no por puntos y comas.

---

[1] Como complemento debe mencionarse que una estructura en C++ también puede ser declarada como una clase sin funciones miembro y todos los miembros de datos públicos. Del mismo modo, una clase en C++ puede ser declarada como una `struct` que tiene todos los miembros de datos privados y todas las funciones miembro públicas. Por tanto, C++ proporciona dos sintaxis tanto para estructuras como para clases. La convención, sin embargo, es no mezclar las notaciones y siempre usar estructuras para crear tipos de registro y clases para proporcionar información verdadera y ocultar la implementación.

Program 13.2

```
#include <iostream>
using namespace std;

struct Date    // this is a global declaration
{
  int month;
  int day;
  int year;
};

int main()
{
  Date birth;

  birth.month = 12;
  birth.day = 28;
  birth.year = 86;

  cout << "My birth date is " << birth.month << '/'
       << birth.day   << '/'
       << birth.year  << endl;
```

Los miembros individuales de una estructura no están restringidos a tipos de datos en números enteros, como lo ilustra la estructura `Fecha`. Puede utilizarse cualquier tipo de datos válido en C++. Por ejemplo, considérese un registro de empleados consistente en los siguientes elementos de datos:

```
Nombre:
Número de identificación:
Tarifa de pago normal:
Tarifa de pago de tiempo extra:
```

Una declaración adecuada para estos elementos de datos es

```
struct Reg_pago
{
  string nombre;
  int num_id;
  double tarifa_normal;
  double tarifa_textra;
};
```

Una vez que se ha declarado el tipo de datos `Reg_pago`, puede definirse e inicializarse una variable de estructura específica usando este tipo. Por ejemplo, la definición

```
Reg_pago empleado = {"H. Price", 12387, 15.89, 25.50};
```

crea una estructura llamada `empleado` del tipo de datos de `Reg_pago`. Los miembros individuales de `empleado` se inicializan con los datos respectivos enlistados entre llaves en la instrucción de definición.

Hay que observar que una estructura sencilla tan sólo es un método conveniente para combinar y almacenar elementos relacionados bajo un nombre común. Aunque una estructura sencilla es útil para identificar de manera explícita la relación entre sus miembros, los miembros individuales podrían definirse como variables separadas. Una de las ventajas reales de usar estructuras sólo se nota cuando el mismo tipo de datos se usa muchas veces en una lista. La creación de listas con el mismo tipo de datos es el tema de la siguiente sección.

Antes de dejar las estructuras sencillas, vale la pena señalar que los miembros individuales de una estructura pueden ser cualquier tipo de datos válido en C++, incluyendo arreglos y estructuras. Se usó un arreglo de caracteres como un miembro de la estructura empleado definida con anterioridad. Para tener acceso a un elemento de un arreglo miembro se requiere dar el nombre de la estructura, seguido por un punto, seguido por la designación del arreglo.

Incluir una estructura dentro de una estructura sigue las mismas reglas para incluir cualquier tipo de datos en una estructura. Por ejemplo, supóngase que una estructura consiste en un nombre y una fecha de nacimiento, donde se ha declarado una estructura `Fecha` como

```
struct Datos
{
  int mes;
  int dia;
  int anio;
};
```

Una definición adecuada de una estructura que incluya un `nombre` y una estructura de `Fecha` es

```
struct
{
struct Datos
  string nombre;
  Fecha nacimiento;
} persona;
```

Hay que observar que al declarar la estructura `Fecha`, el término `Fecha` es el nombre de un tipo de datos; por tanto aparece antes de las llaves en la instrucción de declaración. Al definir la variable de estructura para persona, `persona` es un nombre de variable; por tanto es el nombre de una estructura específica. Lo mismo sucede con la variable llamada `nacimiento`. Éste es el nombre de una estructura `Fecha` específica. Se tiene acceso a los miembros individuales en la estructura `persona` precediendo al miembro deseado con el nombre de la estructura seguido por un punto. Por ejemplo, `persona.nacimiento.mes` se refiere a la variable `mes` en la estructura `nacimiento` contenida en la estructura `persona`.

Ejercicios 13.1

1. Declare un tipo de estructura de datos llamado `Temp_e` para cada uno de los siguientes registros:
   a. el registro de un estudiante que consiste en un número de identificación del estudiante, número de créditos completados y el promedio acumulado de puntos de calificación.
   b. el registro de un estudiante que consiste en el nombre de un estudiante, la fecha de nacimiento, el número de créditos completados y el promedio acumulado de puntos de calificación.
   c. un registro de inventario que consiste en los elementos ilustrados antes en la figura 13.1.
   d. un registro de acciones que consiste en el nombre de la acción, el precio de la acción y la fecha de compra.
   e. un registro de inventario que consiste en un número de parte entero, la descripción de la parte, el número de partes en el inventario y un número entero para reordenar

2. Para los tipos de datos individuales declarados en el ejercicio 1, defina un nombre de variable de estructura adecuado e inicialice cada estructura con los siguientes datos apropiados:

   a. Número de identificación: 4672
      Número de créditos completados: 68
      Promedio de puntos de calificación: 3.01
   b. Nombre: Rhona Karp
      Fecha de nacimiento: 8/4/60
      Número de créditos completados: 96
      Promedio de puntos de calificación: 3.89
   c. Número de parte: 54002
      Familia de CI: ECL
      Tipo de función: NAND
      Suministro de voltaje: –5
      Unidades en existencia: 123
   d. Acción: IBM
      Precio de compra: 134.5
      Fecha de compra: 10/1/86
   e. Número de parte: 16879
      Descripción: Batería
      Número en existencia: 10
      Número de reorden: 3

3. a. Escriba un programa en C++ que pida a un usuario que introduzca el mes, día y año actuales. Almacene los datos introducidos en un registro definido de manera adecuada y despliegue la fecha de una manera apropiada.
   b. Modifique el programa escrito en el ejercicio 3a para usar un registro que acepte el tiempo actual en horas, minutos y segundos.

Point of Information

Homogeneous and Heterogeneous Data Structures

Both arrays and structures are structured data types. The difference between these tures is the types of elements they contain. An array is a *homogeneous* data structu

4. Escriba un programa en C++ que use una estructura para almacenar el nombre de una acción, sus ganancias estimadas por acción y la proporción estimada de costo-utilidad. Haga que el programa pida al usuario que introduzca estos elementos para cinco acciones diferentes, usando cada vez la misma estructura para almacenar los datos introducidos. Cuando se hayan introducido los datos para una acción particular, haga que el programa calcule y despliegue el precio de la acción anticipado basado en los valores de las ganancias y costo-utilidad introducidos. Por ejemplo, si un usuario introdujo los datos XYZ 1.56 12, el precio anticipado para una acción XYZ es (1.56)*(12) = $18.72.

5. Escriba un programa en C++ que acepte un tiempo introducido por el usuario en horas y minutos. Haga que el programa calcule y despliegue el tiempo un minuto después.

6. **a.** Escriba un programa en C++ que acepte una fecha introducida por el usuario. Haga que el programa calcule y despliegue la fecha del día siguiente. Para propósitos de este ejercicio, suponga que todos los meses constan de 30 días.
   b. Modifique el programa escrito en el ejercicio 6a para que dé cuenta del número real de días en cada mes.

13.2 ARREGLOS DE ESTRUCTURAS

El poder verdadero de las estructuras se comprende cuando la misma estructura se usa para listas de datos. Por ejemplo, suponga que los datos mostrados en la figura 13.3 deben ser procesados. Es evidente que los números de empleado pueden almacenarse juntos en un arreglo de números enteros, los nombres en un arreglo de cadenas y las tarifas de pago en un arreglo de números de precisión doble. Al organizar los datos de esta forma, cada columna en la figura 13.3 es considerada como una lista separada, la cual es almacenada en su propio arreglo. La correspondencia entre elementos para cada empleado individual se mantiene almacenando los datos de un empleado en la misma posición en cada arreglo.

Employee Number	Employee Name	Employee Pay Rate
32479	Abrams, B.	6.72
33623	Bohm, P.	7.54
34145	Donaldson, S.	5.56
35987	Ernst, T.	5.43
36203	Gwodz, K.	8.72
36417	Hanson, H.	7.64
37634	Monroe, G.	5.29

Figura 13.3 *Una lista de datos de empleados.*

La separación de la lista completa en tres arreglos individuales es desafortunada, porque todos los elementos que se relacionan con un solo empleado constituyen una organización natural de los datos en estructuras, como se ilustra en la figura 13.4. Usando una estructura, la integridad de la organización de los datos como un registro puede ser mantenida y reflejada por el programa. Bajo este enfoque, la lista ilustrada en la figura 13.4 puede procesarse como un arreglo sencillo de diez estructuras.

	Employee Number	Employee Name	Empl Pay I
1st structure →	32479	Abrams, B.	6.72
2nd structure →	33623	Bohm, P.	7.54
3rd structure →	34145	Donaldson, S.	5.56
4th structure →	35987	Ernst, T.	5.43
5th structure →	36203	Gwodz, K.	8.72
6th structure →	36417	Hanson, H.	7.64
7th structure →	37634	Monroe, G.	5.29

Figura 13.4 *Una lista de estructuras.*

Declarar un arreglo de estructuras es lo mismo que declarar un arreglo de cualquier otro tipo de variable. Por ejemplo, si el tipo de datos `Reg_pago` es declarado como

```
struct Reg_pago {int num_id; string nombre; double tarifa;};
```

entonces un arreglo de diez estructuras de éstas puede definirse como

```
Reg_pago empleado[10];
```

Esta instrucción de definición construye un arreglo de diez elementos, cada uno de los cuales es una estructura del tipo de datos `Reg_pago`. Hay que observar que la creación de un arreglo de diez estructuras tiene la misma forma que la creación de cualquier otro arreglo. Por ejemplo, crear un arreglo de diez números enteros llamada `empleado` requiere la declaración

```
int empleado[10];
```

En esta declaración el tipo de datos es un número entero, mientras en la declaración anterior para empleado el tipo de datos es `Reg_pago`.

Una vez que se ha declarado un arreglo de estructuras, se referencia un elemento de datos particular dando la posición de la estructura deseada en el arreglo seguida por un punto y el miembro de la estructura apropiado. Por ejemplo, la variable `empleado-[0].tarifa` hace referencia al miembro `tarifa` de la primera estructura `empleado` en el arreglo `empleado`. Incluir estructuras como elementos de un arreglo permite que se procese una lista de estructuras usando técnicas de programación de arreglos estándares. El programa 13.3 despliega los registros de los primeros cinco empleados ilustrados en la figura 13.4.

Program 13.3

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

const int NUMRECS = 5;   // maximum number of records

struct PayRec   // this is a global declaration
{
  int id;
  string name;
  double rate;
};

int main()
{
  int i;
  PayRec employee[NUMRECS] = {
                              { 32479, "Abrams, B.", 6.72 },
                              { 33623, "Bohm, P.", 7.54},
                              { 34145, "Donaldson, S.",  5.56},
                              { 35987, "Ernst, T.", 5.43 },
```

(Continued)

```
cout << endl;   // start on a new line
cout << setiosflags(ios::left);  // left justify the output
for ( i = 0; i < NUMRECS; i++)
  cout << setw(7)  << employee[i].id
       << setw(15) << employee[i].name
       << setw(6)  << employee[i].rate << endl;
```

La salida desplegada por el programa 13.3 es:

```
32479  Abrams, B.     6.72
33623  Bohm, P.       7.54
34145  Donaldson, S.  5.56
35987  Ernst, T.      5.43
36203  Gwodz, K.      8.72
```

Al revisar el programa 13.3 se puede observar la inicialización del arreglo de estructuras. Aunque los inicializadores para cada estructura se han encerrado en llaves interiores, éstas no son estrictamente necesarias porque todos los miembros se han inicializado. Como con todas las variables externas y estáticas, en ausencia de inicializadores explícitos, los elementos numéricos de los arreglos o estructuras estáticos y externos se inicializan en cero y sus elementos de carácter se inicializan en `NULL`. El manipulador `setiosflags (ios::left)` incluido en el flujo del objeto `cout` obliga a que cada nombre sea desplegado justificado a la izquierda en su ancho de campo designado.

Ejercicios 13.2

1. Defina arreglos de 100 estructuras para cada uno de los tipos de datos descritos en el ejercicio 1 de la sección anterior.
2. **a.** Usando el tipo de datos

```
struct MesDias
{
  string nombre;
  int dias;
};
```

defina un arreglo de 12 estructuras del tipo `MesDias`. Nombre al arreglo `convertir[]` e inicialice el arreglo con los nombres de los 12 meses en un año y el número de días en cada mes.

b. Incluya el arreglo creado en el ejercicio 2a en un programa que despliegue los nombres y número de días en cada mes.

3. Usando el tipo de datos declarado en el ejercicio 2a, escriba un programa en C++ que acepte un mes de un usuario en forma numérica y despliegue el nombre del mes y el número de días en el mes. Por tanto, en respuesta a una entrada de 3, el programa desplegará `Marzo tiene 31 dias`.

4. a. Declare una estructura sencilla con un tipo de datos adecuado para una estructura de empleados del tipo ilustrado a continuación:

Number	Name	Rate	I
3462	Jones	4.62	4
6793	Robbins	5.83	3
6985	Smith	5.22	4

b. Usando el tipo de datos declarado en el ejercicio 4a, escriba un programa en C++ que acepte de manera interactiva los datos anteriores en un arreglo de seis estructuras. Una vez que se han introducido los datos, el programa deberá crear un reporte de nómina enlistando el nombre, número y salario bruto de cada empleado. Incluya el pago bruto total de todos los empleados al final del reporte.

5. a. Declare una estructura de datos sencilla con un tipo de datos adecuado para una estructura de automóvil del tipo ilustrado:

Car Number	Miles Driven	Gallons Used
25	1450	62
36	3240	136

b. Usando el tipo de datos declarado para el ejercicio 5a, escriba un programa en C++ que acepte en forma interactiva los datos anteriores en un arreglo de cinco estructuras. Una vez que se hayan introducido los datos, el programa deberá crear un reporte mostrando el número de automóvil y las millas por galón logradas por cada automóvil. Al final del reporte incluya las millas por galón promedio logradas por la flotilla completa de automóviles.

13.3 ESTRUCTURAS COMO ARGUMENTOS DE FUNCIÓN

Los miembros individuales de estructura pueden transmitirse a una función de la misma manera que cualquier variable escalar. Por ejemplo, dada la definición de estructura

```
struct Empleado
{
  int num_id;
  double tarifaPago;
  double horas;
} empl;
```

la instrucción

```
desplegar(empl.num_id);
```

transmite una copia del miembro de la estructura `empl.num_id` a una función llamada `desplegar()`. Del mismo modo, la instrucción

```
calcPago(empl.tarifaPago,empl.horas);
```

transmite copias de los valores almacenados en los miembros de la estructura `empl.tarifaPago` y `empl.horas` a la función `calcPago()`. Ambas funciones, `desplegar()` y `calcPago()`, deben declarar los tipos de datos correctos para sus respectivos argumentos.

También pueden transmitirse copias completas de todos los miembros de una estructura a una función al incluir el nombre de la estructura como un argumento para la función llamada. Por ejemplo, la llamada a la función

```
calcNeto(empl);
```

transmite una copia de la estructura `empl` completa a `calcNeto()`. Dentro de `calcNeto()` debe hacerse una declaración apropiada para recibir la estructura. El programa 13.4 declara un tipo de datos global para una estructura de empleados. Este tipo es usado entonces por las funciones `main()` y `calcNeto()` para definir estructuras específicas con los `nombres empl` y `temp`, respectivamente.

La salida producida por el programa 13.4 es:

```
El pago neto para el empleado 6782 es $361.66
```

Al revisar el programa 13.4, se observará que tanto `main()` como `calcNeto()` usan el mismo tipo de datos para definir sus variables individuales de estructura. La variable de estructura definida en `main()` y la variable de estructura definida en `calcNeto()` son dos estructuras completamente diferentes. Los cambios hechos a la variable local `temp` en `calcNeto()` no se reflejan en la variable `empl` de `main()`. De hecho, dado que ambas variables de estructura son locales para sus respectivas funciones, podría haberse usado el mismo nombre de variable de estructura en ambas funciones sin ambigüedad.

Programa 13.4

```
#include <iostream>
#include <iomanip>
using namespace std;

struct Empleado        // declara un tipo global
{
  int num_id;
  double tarifaPago;
  double horas;
};

double calcNeto(Empleado);     // prototipo de la funcion

int main()
{
  Empleado empl = {6782, 8.93, 40.5};
  double pagoNeto;

  pagoNeto = calcNeto(empl);     // transmite copias de los valores en empl
    // establece formatos de salida
  cout << setw(10)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);

  cout << "El pago neto para el empleado " << empl.num_id
       << " es $" << pagoNeto << endl;

  return 0;
}

double calcNeto(Empleado temp) // temp es del tipo de datos Empleado
{
  return temp.tarifaPago * temp.horas;
}
```

Cuando `calcNeto()` es llamado por `main()`, se transmiten copias de los valores de la estructura `empl` a la estructura `temp`. Entonces `calcNeto()` usa dos de los valores miembro transmitidos para calcular un número, el cual es devuelto a `main()`. Dado que `calcNeto()` devuelve un número que no es entero, el tipo de datos del valor devuelto debe incluirse en todas las declaraciones para `calcNeto()`.

Una alternativa para la llamada a la función con transmisión por valor ilustrada en el programa 13.4, en el cual la función llamada recibe una copia de una estructura, es una transmisión por referencia, la que transmite una referencia a una estructura. Hacerlo así permite que la función llamada tenga acceso directo a la variable de estructura en la fun-

ción que llama y alterar sus valores. Por ejemplo, haciendo referencia al programa 13.4, el prototipo de `calcNeto()` puede modificarse a

```
double calcNeto(Empleado &);
```

Si se usa este prototipo de la función y la función `calcNeto()` se vuelve a escribir para ajustarse a él, la función `main()` en el programa 13.4 puede usarse como está. El programa 13.4a ilustra estos cambios dentro del contexto de un programa completo.

Program 13.4a

```
#include <iostream>
#include <iomanip>
using namespace std;

struct Employee       // declare a global type
{
  int idNum;
  double payRate;
  double hours;
};

double calcNet(Employee&);   // function prototype

int main()
{
  Employee emp = {6782, 8.93, 40.5};
  double netPay;

  netPay = calcNet(emp);       // pass a reference

    // set output formats
  cout << setw(10)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);

  cout << "The net pay for employee " << emp.idNum
       << " is $" <<  netPay << endl;

  return 0;
}

double calcNet(Employee& temp)   // temp is a reference variable
{
```

El programa 13.4a produce la misma salida que el programa 13.4, excepto que la función `calcNeto()` en el programa 13.4a recibe acceso directo a la estructura `empl` en lugar de una copia de ellas. Esto significa que el nombre de la variable `temp` dentro de `calcNeto` es un nombre alterno para la variable `empl` en `main()` y los cambios a `temp` son cambios directos a `empl`. Aunque en ambos programas se hace la misma llamada a la función, `calcNeto(empl)`, la llamada en el programa 13.4a transmite una referencia, mientras que en el programa 13.4 transmite valores.

Transmisión de un apuntador

En lugar de transmitir una referencia, puede usarse un apuntador. Usar un apuntador requiere, además de modificar el prototipo de la función y la línea de encabezado, que la llamada a `calcNeto()` en el programa 13.4 se modifique a

```
calcNeto(&empl);
```

Aquí la llamada a la función indica con claridad que se está transmitiendo una dirección (lo cual no sucede en el programa 13.4a). La desventaja, sin embargo, está en la notación de desreferenciación requerida en forma interna en la función. No obstante, como los apuntadores se usan en forma amplia en la práctica, vale la pena familiarizarse con la notación empleada.

Para almacenar en forma correcta la dirección transmitida, `calcNeto()` debe declarar sus parámetros como un apuntador. Una definición de función adecuada para `calcNeto()` es

```
calcNeto(Empleado *pt)
```

Aquí, la declaración para `pt` declara este parámetro como un apuntador a una estructura de tipo `Empleado`. El apuntador `pt` recibe la dirección inicial de una estructura siempre que se llame a `calcNeto()`. Dentro de `calcNeto()`, este apuntador se usa para hacer referencia directa a cualquier miembro en la estructura. Por ejemplo `(*pt).num_id` se refiere al miembro `num_id` de la estructura `(*pt).tarifaPago` se refiere al miembro `tarifaPago` de la estructura y `(*pt).horas` se refiere al miembro `horas` de la estructura. Estas relaciones se ilustran en la figura 13.5.

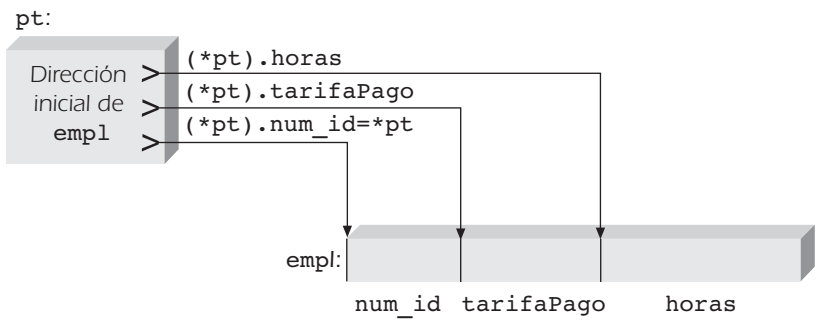


Figura 13.5 *Se puede utilizar un apuntador para tener acceso a los miembros de la estructura.*

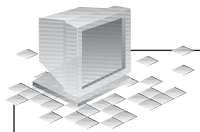
Los paréntesis que rodean a la expresión `*pt` en la figura 13.5 son necesarios para tener acceso inicial a "la estructura cuya dirección está en `pt`". Esto es seguido por un identificador para tener acceso al miembro deseado dentro de la estructura. En ausencia de paréntesis, el operador `.` del miembro de la estructura toma precedencia sobre el operador de indirección. Por tanto, la expresión `*pt.horas` es otra forma de escribir `*(pt.horas)`, lo cual se referiría a "la variable cuya dirección está en la variable `pt.horas`". Es evidente que esta última expresión no tiene sentido porque no hay una estructura llamada `pt` y `horas` no contiene una dirección. Como se ilustra en la figura 13.5, la dirección inicial de la estructura `empl` también es la dirección del primer miembro de la estructura.

El uso de apuntadores de esta manera es tan común que existe una notación especial para ello. La expresión general `(*apuntador).miembro` siempre puede reemplazarse con la notación `apuntador->miembro`, donde el operador `->` se construye usando un signo de menos seguido por una flecha que apunta a la derecha (símbolo mayor que). Puede usarse cualquier expresión para localizar el miembro deseado. Por ejemplo, las siguientes expresiones son equivalentes:

`(*pt).num_id`	puede reemplazarse con	`pt->num_id`
`(*pt).tarifaPago`	puede reemplazarse con	`pt->tarifaPago`
`(*pt).horas`	puede reemplazarse con	`pt->horas`

El programa 13.5 ilustra la transmisión de la dirección de una estructura y el uso de un apuntador con la nueva notación para hacer referencia directa a la estructura.

El nombre del parámetro apuntador declarado en el programa 13.5 es seleccionado, por supuesto, por el programador. Cuando se llama a `calcNeto()`, la dirección inicial de `empl` es transmitida a la función. Usando esta dirección como punto de partida, se tiene acceso a los miembros individuales de la estructura al incluir sus nombres con el apuntador.



Program 13.5

```
#include <iostream>
#include <iomanip>
using namespace std;

struct Employee   // declare a global type
{
  int idNum;
  double payRate;
  double hours;
};

double calcNet(Employee *);   //function prototype

int main()
{
  Employee emp = {6782, 8.93, 40.5};
  double netPay;
  netPay = calcNet(&emp);       // pass an address

    // set output formats
  cout << setw(10)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);

  cout << "The net pay for employee " << emp.idNum
       << " is $" << netPay << endl;

  return 0;
}

double calcNet(Employee *pt)   // pt is a pointer to a
{                                 // structure of Employee type
```

Como con todas las expresiones en C++ que tienen acceso a una variable, también pueden aplicarse a ellas los operadores de incremento y decremento. Por ejemplo, la expresión

```
++pt->horas
```

añade uno al miembro `horas` de la estructura `empl`. Dado que el operador `->` tiene una prioridad mayor que el operador de incremento, primero se tiene acceso al miembro `horas` y luego se aplica el incremento. De manera alternativa, la expresión `(++pt)->horas` usa el prefijo del operador de incremento para aumentar la dirección en `pt` antes que se tenga acceso al miembro `horas`. Del mismo modo, la expresión `(pt++)->horas` usa el posfijo del operador de incremento para aumentar la dirección en `pt` después que se tie-

ne acceso al miembro `horas`. En ambos casos, sin embargo, debe haber suficientes estructuras definidas para asegurar que los apuntadores incrementados en realidad apuntan a estructuras legítimas.

Como ejemplo, la figura 13.6 ilustra un arreglo de tres estructuras de tipo `empleado`. Suponiendo que la dirección de `empl[1]` está almacenada en la variable apuntadora `pt`, la expresión `++pt` cambia la dirección en `pt` a la dirección inicial de `empl[2]`, mientras la expresión `--pt` cambia la dirección para apuntar a `empl[0]`.

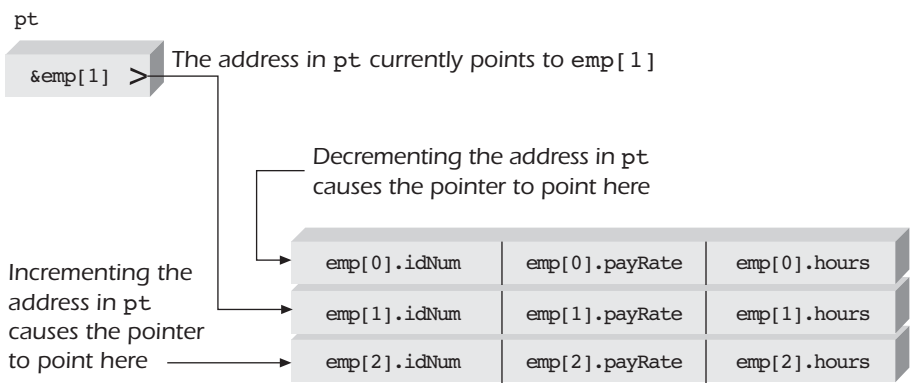


Figura 13.6 *Cambiar direcciones en el apuntador.*

Devolución de estructuras

En la práctica, la mayor parte de las funciones que manejan estructuras reciben acceso directo a una estructura al recibir una referencia o dirección a una estructura. Entonces pueden hacerse los cambios a la estructura en forma directa desde dentro de la función. Sin embargo, si se desea hacer que una función devuelva una estructura separada, deben seguirse los mismos procedimientos para devolver estructuras de datos completas que para devolver valores escalares. Estos procedimientos incluyen declarar la función de manera apropiada y alertar a cualquier función que llame del tipo de estructura de datos que se está devolviendo. Por ejemplo, la función `obtenerValores()` en el programa 13.6 devuelve una estructura completa a `main()`.

Program 13.6

```
#include <iostream>
#include <iomanip>
using namespace std;

struct  Employee       // declare a global type
{
  int idNum;
  double payRate;
  double hours;
};

Employee getVals();   // function prototype

int main()
{
  Employee emp;

  emp = getVals();
  cout << "\nThe employee id number is " << emp.idNum
       << "\nThe employee pay rate is $" << emp.payRate
       << "\nThe employee hours are " << emp.hours << endl;

  return 0;
}

Employee getVals() // return an employee structure
{
  Employee next;

  next.idNum = 6789;
  next.payRate = 16.25;
  next.hours = 38.0;
```

La siguiente salida se despliega cuando se ejecuta el programa 13.6:

```
El número de identificación del empleado es 6789
La tarifa de pago del empleado es $16.25
Las horas del empleado son 38
```

Dado que la función `obtenerValores()` devuelve una estructura, el encabezado de la función para `obtenerValores()` debe especificar el tipo de estructura que se está devolviendo. Debido a que `obtenerValores()` no recibe ningún argumento, el encabezado de la función no tiene declaraciones de parámetros y consiste en la línea

```
Empleado obtenerValores();
```

Dentro de `obtenerValores()`, la variable `siguiente` se define como una estructura del tipo que se va a devolver. Después que se han asignado valores a la estructura `siguiente`, los valores de la estructura son devueltos al incluir el nombre de la estructura dentro de los paréntesis de la instrucción de devolución.

En el lado receptor, debe alertarse a `main()` que la función `obtenerValores()` devolverá una estructura. Esto se maneja incluyendo una declaración de función para `obtenerValores()` en `main()`. Hay que observar que estos pasos para devolver una estructura a partir de una función son idénticos a los procedimientos normales para devolver tipos de datos escalares descritos antes en el capítulo 6.

Ejercicios 13.3

1. Escriba una función en C++ llamada `dias()` que determine el número de días desde el inicio del siglo para cualquier fecha transmitida como una estructura. Use la estructura `Fecha`

```
struct Fecha
{
  int mes;
  int dia;
  int anio;
};
```

   Al escribir la función `dias()` use la convención de que todos los años tienen 360 días y cada mes consiste en 30 días. La función deberá devolver el número de días para cualquier estructura `Fecha` que se le transmita.

2. Escriba una función en C++ llamada `dif_dias()` que calcule y devuelva la diferencia entre dos fechas. Cada fecha es transmitida a la función como una estructura usando el siguiente tipo global:

```
struct Fecha
{
  int mes;
  int dia;
  int anio;
};
```

   La función `dif_dias()` deberá hacer dos llamadas a la función `dias()` escrita para el ejercicio 1.

3. **a.** Vuelva a escribir la función `dias()` redactada para el ejercicio 1 para recibir una referencia a una estructura `Fecha`, en lugar de una copia de la estructura completa.
   b. Repita el ejercicio 3a usando un apuntador en lugar de una referencia.

4. **a.** Escriba una función en C++ llamada `mayor()` que devuelva la fecha más reciente de cualquier par de fechas que se le transmitan. Por ejemplo, si se transmiten las fechas 10/9/2005 y 11/3/2005 a `mayor()`, será devuelta la segunda fecha.
   b. Incluya la función `mayor()` que se escribió para el ejercicio 4a en un programa completo. Almacene la estructura `Fecha` devuelta por `mayor()` en una estructura `Fecha` separada y despliegue los valores miembros de la `Fecha` devuelta.

5. a. En dos dimensiones, un vector matemático es un par de números que representan flechas dirigidas en un plano, como lo muestran los vectores matemáticos v_1 y v_2 en la figura 13.7.

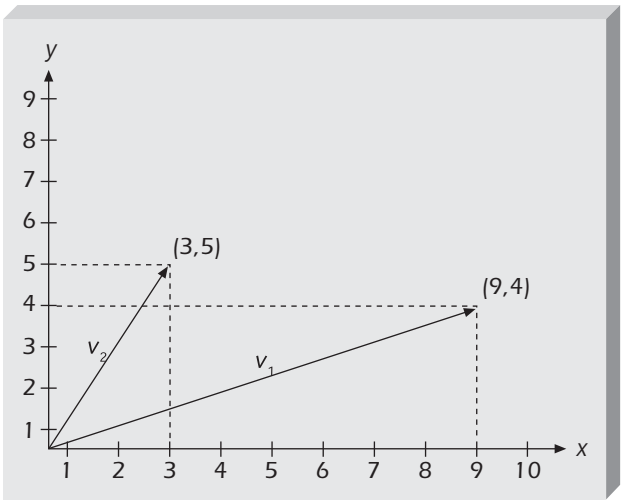


Figura 13.7 Gráfica para el ejercicio 5.

Los vectores matemáticos bidimensionales pueden escribirse en la forma (a,b), donde a y b son llamados los componentes x y y del vector, respectivamente. Por ejemplo, para los vectores ilustrados en la figura 13.7, $v_1 = (9, 4)$ y $v_2 = (3, 5)$. Para los vectores se aplican las siguientes operaciones:

Si $v_1 = (a,b)$ y $v_2 = (c,d)$
$v_1 + v_2 = (a,b) + (c,d) = (a + c, b + d)$
$v_1 - v_2 = (a,b) - (c,d) = (a - c, b - d)$

Usando esta información, escriba un programa en C++ que defina un arreglo de dos registros de vectores, donde cada registro consista en dos componentes de precisión doble a y b. Su programa deberá permitir a un usuario introducir dos vectores, llamar a dos funciones que devuelvan la suma y la diferencia de los vectores introducidos, y desplegar los resultados calculados por estas funciones.

b. Además de las operaciones definidas en el ejercicio 5a, dos operaciones adicionales con vectores son la negación y el valor absoluto. Para un vector v_1 con componentes (a,b) estas operaciones se definen como sigue:

negación: $-v_1 = -(a,b) = (-a,-b)$
valor absoluto: $| v_1 |$ = sqrt $(a * a + b * b)$

Usando esta información, modifique el programa que escribió para el ejercicio 5a para desplegar la negación y los valores absoluto de ambos vectores introducidos por un usuario al igual que la negación y el valor absoluto de la suma de los dos vectores introducidos.

13.4 LISTAS VINCULADAS

Un problema clásico del manejo de datos es hacer adiciones o eliminaciones a estructuras existentes que se mantienen en un orden específico. Esto se ilustra mejor al considerar la lista alfabética de teléfonos que se muestra en la figura 13.8. Empezando con este conjunto inicial de nombres y números telefónicos, se desea agregar estructuras nuevas a la lista en la secuencia alfabética apropiada, y eliminar estructuras existentes, de tal forma que se elimine el almacenamiento para las estructuras eliminadas.

Amador, Samuel
(555) 898-2392
Domínguez, Edith
(555) 682-3104
López, Juan
(555) 718-4581
Martínez, Esteban
(555) 382-7070
Zapata, Horacio
(555) 219-9912

Figura 13.8 *Una lista telefónica en orden alfabético.*

Aunque la inserción o eliminación de estructuras ordenadas puede lograrse usando un arreglo de estructuras, estos arreglos no son representaciones eficientes para agregar o eliminar estructuras internas en el arreglo. Los arreglos son fijos y tienen un tamaño especificado con anterioridad. Eliminar una estructura de un arreglo crea un espacio vacío que requiere una marcación especial o mover hacia arriba todos los elementos debajo de la estructura eliminada para cerrar el espacio vacío. Del mismo modo, agregar una estructura al cuerpo de un arreglo de estructuras requiere que todos los elementos debajo de la adición se muevan hacia abajo para hacer espacio a la nueva entrada, o el nuevo elemento podría agregarse al final del arreglo existente y luego tendría que reordenarse el arreglo para restablecer el orden apropiado de las estructuras. Por tanto, agregar o eliminar registros de una lista así por lo general requiere reestructurar y volver a escribir la lista, una práctica molesta, tardada e ineficiente.

Una lista vinculada proporciona un método conveniente para mantener una lista que cambia en forma constante, sin necesidad de reordenar y reestructurar de manera continua la lista completa. Una lista vinculada es tan sólo un conjunto de estructuras en el que cada estructura contiene al menos un miembro cuyo valor es la dirección de la siguiente estructura ordenada lógicamente en la lista. En lugar de requerir que cada registro esté almacenado en forma física en el orden apropiado, cada estructura nueva se agrega de manera física siempre que la computadora tiene espacio libre en su área de almacenamiento. Los registros se "vinculan" incluyendo la dirección del siguiente registro en el registro que lo precede inmediatamente. Desde el punto de vista de la programación, la estructura actual que se está procesando contiene la dirección del siguiente registro, sin importar dónde esté almacenada en realidad la siguiente estructura.

El concepto de lista vinculada se ilustra en la figura 13.9. Aunque los datos reales para la estructura `López` ilustrada en la figura pueden estar almacenados físicamente en cualquier parte en la computadora, el miembro adicional incluido al final de la estructura `Domínguez` mantiene el orden alfabético apropiado. Este miembro proporciona la dirección inicial de la ubicación donde está almacenado el registro López. Como podría esperarse, este miembro es un apuntador.

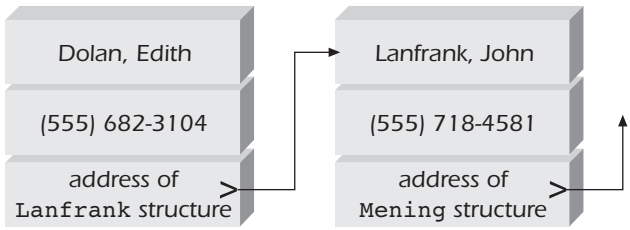


Figura 13.9 *Uso de apuntadores para vincular estructuras.*

Para poder observar la utilidad del apuntador en la estructura Domínguez, se agregará un número telefónico para Julia Hernández en la lista alfabética mostrada en la figura 13.8. Los datos para Julia Hernández se almacenan en una estructura de datos empleando el mismo tipo utilizado para las estructuras existentes. Para asegurar que el número telefónico para Hernández se despliega de manera correcta después del número telefónico para Domínguez, la dirección en la estructura Domínguez debe alterarse para que apunte a la estructura Hernández, y la dirección en la estructura Hernández debe establecerse para que apunte a la estructura López. Esto se ilustra en la figura 13.10. Hay que observar que el apuntador en cada estructura tan sólo apunta a la ubicación de la siguiente estructura ordenada, aun si esa estructura no se localiza físicamente en el orden correcto.

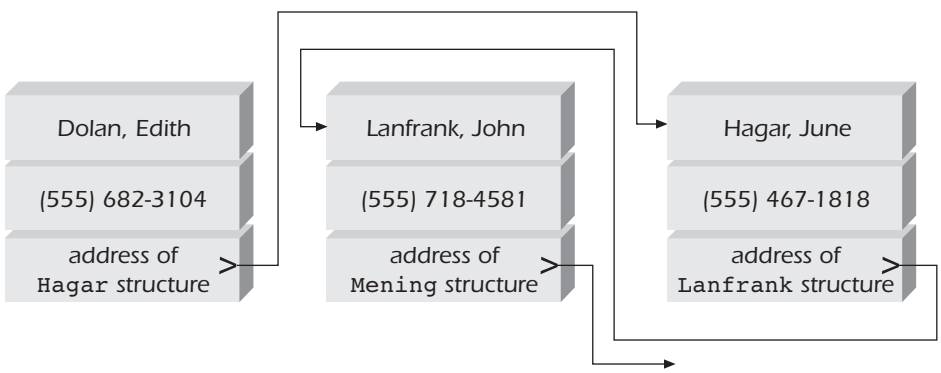


Figura 13.10 *Ajustar direcciones para que apunten a estructuras apropiadas.*

La eliminación de una estructura de la lista ordenada es el proceso inverso de agregar un registro. El registro real se elimina lógicamente de la lista con sólo cambiar la dirección en la estructura que la precede para que apunte a la que sigue de inmediato al registro eliminado.

Cada estructura en una lista vinculada tiene el mismo formato; sin embargo, es claro que el último registro no puede tener un valor apuntador válido que apunte a otro registro, debido a que no hay ninguno. C++ proporciona un valor apuntador especial llamado `NULL` que actúa como un centinela o bandera para indicar cuando se ha procesado el último registro. El valor apuntador `NULL`, como su contraparte de fin de cadena, tiene un valor numérico de cero.

Además de un valor centinela de fin de lista, también debe proporcionarse un apuntador especial para almacenar la dirección de la primera estructura en la lista. La figura 13.11 ilustra el conjunto completo de apuntadores y estructuras para una lista consistente en tres nombres.

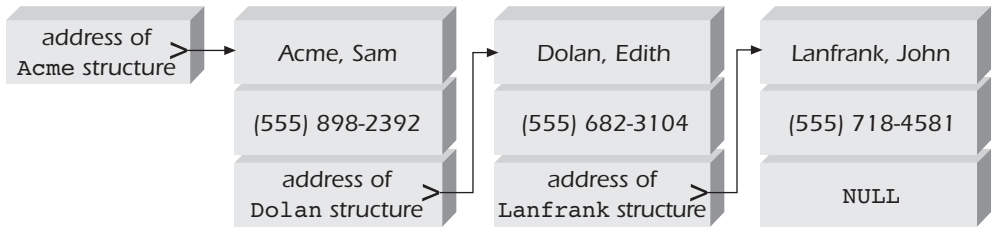


Figura 13.11 *Uso de los valores apuntadores inicial y final.*

La inclusión de un apuntador en una estructura no debería ser sorprendente. Como se descubrió en la sección 13.1, en C++ una estructura puede contener cualquier tipo de datos. Por ejemplo, la declaración de estructura

```
struct Prueba
{
  int num_id;
  double *ptPago
};
```

declara un tipo de estructura consistente en dos miembros. El primer miembro es una variable en número entero llamada `num_id` y la segunda variable es un apuntador llamado `ptPago`, el cual es un apuntador a un número de precisión doble. El programa 13.7 ilustra que el miembro apuntador de una estructura se usa como cualquier otra variable apuntadora.

La salida producida al ejecutar el programa 13.7 es

```
El empleado numero 12345 recibio un pago de $456.20
```

La figura 13.12 ilustra la relación entre los miembros de la estructura `empl` definida en el programa 13.7 y la variable llamada pago. El valor asignado a `empl.num_id` es el número 12345 y el valor asignado a pago es 456.20. La dirección de la variable `pago` es asignada al miembro de la estructura `empl.ptPago`. Dado que este miembro se ha definido como un apuntador a un número de precisión doble, colocar la dirección de la variable de precisión doble `pago` en él es un uso correcto de este miembro. Por último, dado que el operador de miembro `.` tiene una precedencia mayor que el operador de indirección `*`, la expresión usada en la instrucción `cout` en el programa 13.7 es correcta. La expresión `*empl.ptPago` es equivalente a la expresión `*(empl.ptPago)`, la cual se traduce como "la variable cuya dirección está contenida en el miembro `empl.ptPago`".

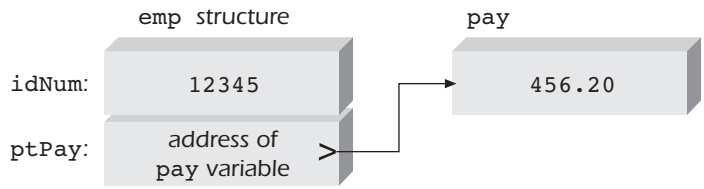


Figura 13.12 *Almacenamiento de una dirección en un miembro de una estructura.*

Program 13.7

```
#include <iostream>
#include <iomanip>
using namespace std;

struct Test
{
  int idNum;
  double *ptPay;
};

int main()
{
  Test emp;
  double pay = 456.20;

  emp.idNum = 12345;
  emp.ptPay = &pay;
    // set output formats
  cout << setw(6)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(2);

  cout << "\nEmployee number " << emp.idNum << " was paid $"
       << *emp.ptPay << endl;
```

Aunque el apuntador definido en el programa 13.7 se ha usado en una forma bastante trivial, el programa ilustra el concepto de incluir un apuntador en una estructura. Este concepto puede extenderse con facilidad para crear una lista vinculada de estructuras adecuada para almacenar los nombres y números telefónicos mostrados en la figura 13.8. La siguiente declaración crea un tipo para una estructura así:

```
struct Tipo_tel
{
  string nombre;
  string num_telef;
  Tipo_tel *siguientedir;
};
```

El último miembro en esta estructura es un apuntador adecuado para almacenar la dirección de una estructura del tipo `Tipo_tel`.

El programa 13.8 ilustra el uso del tipo `Tipo_tel` al definir de manera específica tres estructuras que tienen esta forma. Las tres estructuras se nombran `t1, t2` y `t3`, respectivamente, y los miembros de nombre y número telefónico de cada una de estas estructuras se inicializan cuando se definen las estructuras, usando los datos mostrados en la figura 13.8.

Program 13.8

```
#include <iostream>
#include <string>
using namespace std;

struct TeleType
{
  string name;
  string phoneNo;
  TeleType *nextaddr;
};

int main()
{
  TeleType t1 = {"Acme, Sam","(555) 898-2392"};
  TeleType t2 = {"Dolan, Edith","(555) 682-3104"};
  TeleType t3 = {"Lanfrank, John","(555) 718-4581"};
  TeleType *first;     // create a pointer to a structure

  first = &t1;         // store t1's address in first
  t1.nextaddr = &t2;   // store t2's address in t1.nextaddr
  t2.nextaddr = &t3;   // store t3's address in t2.nextaddr
  t3.nextaddr = NULL;  // store a NULL address in t3.nextaddr

  cout << endl << first->name
       << endl << t1.nextaddr->name
       << endl << t2.nextaddr->name
       << endl;
```

La salida producida al ejecutar el programa 13.8 es

```
Amador, Samuel
Domínguez, Edith
López, Juan
```

El programa 13.8 demuestra el uso de apuntadores para tener acceso a miembros de estructura sucesivos. Como se ilustra en la figura 13.13, cada estructura contiene la dirección de la siguiente estructura en la lista.

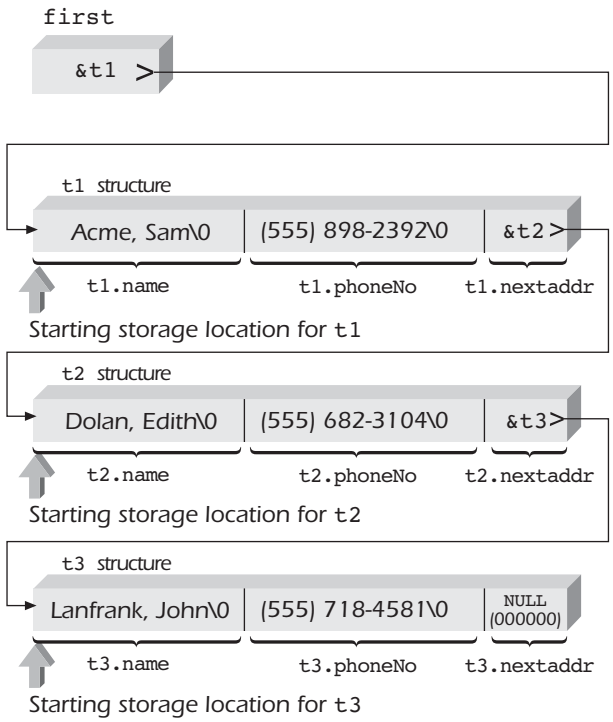


Figura 13.13 *La relación entre estructuras en el programa 13.8.*

La inicialización de los nombres y números telefónicos para cada una de las estructuras definidas en el programa 13.9 es sencilla. Aunque cada estructura consiste en tres miembros, sólo se inicializan los primeros dos miembros de cada estructura. Como ambos miembros son arreglos de caracteres, pueden inicializarse con cadenas. El miembro restante de cada estructura es un apuntador. Para crear una lista vinculada, cada apuntador a la estructura debe tener asignada la dirección de la siguiente estructura en la lista.

Las cuatro instrucciones de asignación en el programa 13.8 ejecutan las asignaciones correctas. La expresión `primero = &t1` almacena la dirección de la primera estructura en la lista en la variable apuntadora llamada primero. La expresión `t1.siguientedir = &t2` almacena la dirección inicial de la estructura `t2` en el miembro apuntador de la estructura `t1`. Del mismo modo, la expresión `t2.siguientedir = &t3` almacena la dirección inicial de la estructura `t3` en el miembro apuntador de la estructura `t2`. Para terminar la lista, el valor del apuntador `NULL`, el cual es cero, se almacena en el miembro apuntador de la estructura `t3`.

Una vez que se han asignado valores a cada miembro de la estructura y se han almacenado direcciones correctas en los apuntadores apropiados, las direcciones en los apuntadores se usan para tener acceso al miembro nombre de cada estructura. Por ejemplo, la expresión `t1.siguientedir->nombre` se refiere al miembro nombre de la estructura cuya dirección está en el miembro `siguientedir` de la estructura `t1`. Las precedencias del operador de miembro y el operador de apuntador de estructura `->` son iguales y se evalúan de izquierda a derecha. Por tanto, la expresión `t1.siguientedir->nombre` se evalúa como `(t1.siguientedir)->nombre`. Dado que `t1.siguientedir` contiene la dirección de la estructura `t2`, se tiene acceso al nombre apropiado.

La expresión `t1.siguientedir->nombre` puede reemplazarse, por supuesto, por la expresión equivalente `(*t1.siguientedir).nombre`, la cual usa el operador de indirección más convencional. Esta expresión también se refiere a "el miembro nombre de la variable cuya dirección está en `t1.siguientedir`".

Las direcciones en una lista de estructuras vinculada pueden usarse para recorrer en un ciclo la lista completa. Conforme se tiene acceso a cada estructura puede examinarse para seleccionar un valor específico o para imprimir una lista completa. Por ejemplo, la función `desplegar()` en el programa 13.9 ilustra el uso de un ciclo `while`, el cual usa la dirección en cada miembro apuntador de la estructura para recorrer en un ciclo toda la lista y desplegar en forma sucesiva los datos almacenados en cada estructura.

La salida producida por el programa 13.9 es

```
Amador, Samuel          (555) 898-2392
Domínguez, Edith        (555) 682-3104
López, Juan             (555) 718-4581
```

El concepto importante ilustrado por el programa 13.9 es el uso de la dirección en una estructura para tener acceso a los miembros de la siguiente estructura en la lista. Cuando se invoca a la función `desplegar()`, se le transmite el valor almacenado en la variable llamada `primero`. Dado que `primero` es una variable apuntadora, el valor real que se transmite es una dirección (la dirección de la estructura `t1`). La función `desplegar()` acepta el valor transmitido en el argumento llamado `contenido`. Para almacenar en forma correcta la dirección transmitida, `contenido` se declara como un apuntador a una estructura del tipo `Tipo_tel`. Dentro de `desplegar()`, se usa un ciclo `while` para recorrer las estructuras vinculadas, empezando con la estructura cuya dirección está en `contenido`. La condición probada en la instrucción `while` compara el valor en `contenido`, el cual es una dirección, con el valor `NULL`. Para cada dirección válida se desplegarán los miembros de nombre y número telefónico de la estructura direccionada. La dirección en `contenido` se actualiza entonces con la dirección en el miembro apuntador de la estructura actual. Entonces se vuelve a evaluar la dirección en `contenido`, y el proceso continúa mientras la dirección en contenido no es igual al valor `NULL`. La función `desplegar()` no "sabe" nada respecto a los nombres de las estructuras declaradas en `main()` o ni siquiera cuántas estructuras existen. Tan sólo recorre la lista vinculada, estructura por estructura, hasta que encuentra la dirección `NULL` de fin de la lista. Dado que el valor de `NULL` es cero, la condición probada puede ser reemplazada por la expresión equivalente `contenido`.

Una desventaja del programa 13.9 es que se definen exactamente tres estructuras en `main()` por nombre y el almacenamiento para ellas se reserva en tiempo de compilación. Si se requiriera una cuarta estructura, la estructura adicional tendría que ser declarada y volver a compilar el programa. En la siguiente sección se muestra cómo hacer que la computadora asigne y libere en forma dinámica almacenamiento para estructuras en tiempo de ejecución, conforme se requiera el almacenamiento. Sólo cuando se va a agregar una estructura nueva a la lista, y mientras el programa esté en ejecución, se crea almacenamiento para la estructura nueva. Del mismo modo, cuando ya no se necesita una estructura y puede eliminarse de la lista, se renuncia al almacenamiento para el registro eliminado y se devuelve a la computadora.

Program 13.9

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct TeleType
{
  string name;
  string phoneNo;
  TeleType *nextaddr;
};

void display(TeleType *);       // function prototype

int main()
{
  TeleType t1 = {"Acme, Sam","(555) 898-2392"};
  TeleType t2 = {"Dolan, Edith","(555) 682-3104"};
  TeleType t3 = {"Lanfrank, John","(555) 718-4581"};
  TeleType *first;    // create a pointer to a structure

  first = &t1;        // store t1's address in first
  t1.nextaddr = &t2;  // store t2's address in t1.nextaddr
  t2.nextaddr = &t3;  // store t3's address in t2.nextaddr
  t3.nextaddr = NULL; // store the NULL address in t3.nextaddr

  display(first);     // send the address of the first structure

  return 0;
}

void display(TeleType *contents) // contents is a pointer to a stru
{                                 // of type TeleType
  while (contents != NULL)   // display till end of linked list
  {
    cout << endl << setiosflags(ios::left)
         << setw(30) << contents->name
         << setw(20) << contents->phoneNo ;
    contents = contents->nextaddr;     // get next address
  }
  cout << endl;

  return;
```

Ejercicios 13.4

1. Modifique el programa 13.9 para solicitar al usuario un nombre. Haga que el programa busque el nombre introducido en la lista existente. Si el nombre está en la lista, despliegue el número telefónico correspondiente; de lo contrario despliegue este mensaje:
   `El nombre no está en el directorio telefónico actual`
2. Escriba un programa en C++ que contenga una lista vinculada de diez números enteros. Haga que el programa despliegue los números en la lista.
3. Usando la lista de estructuras vinculada ilustrada en la figura 13.13, escriba la secuencia de pasos necesaria para eliminar de la lista el registro de Edith Domínguez.
4. Generalice la descripción obtenida en el ejercicio 3 para describir la secuencia de pasos necesaria para eliminar la *n*-ésima estructura de una lista de estructuras vinculadas. La *n*-ésima estructura es precedida por la estructura $(n - 1)$ y es seguida por la estructura $(n + 1)$. Asegúrese de almacenar en forma correcta todos los valores apuntadores.
5. **a.** Una lista con vinculación doble es una lista en la que cada estructura contiene un apuntador tanto a la estructura siguiente como a la anterior en la lista. Defina un tipo apropiado para una lista con doble vinculación de nombres y números telefónicos.
   b. Usando el tipo definido en el ejercicio 5a, modifique el programa 13.9 para enlistar los nombres y números telefónicos en orden inverso.

13.5 ASIGNACIÓN DINÁMICA DE ESTRUCTURAS DE DATOS

Ya se ha encontrado el concepto de asignar y desasignar espacio de memoria en forma explícita usando los operadores new y delete (véase la sección 12.2). Por comodidad se repite la descripción de estos operadores en la tabla 13.1.

Tabla 13.1

Operator Name	Description
`new`	Reserves the number of bytes required by the requested data type. Returns the address of the first reserved location or `NULL`

Esta asignación dinámica de memoria es útil en especial cuando se aborda una lista de estructuras debido a que permite expandir la lista conforme se agregan nuevos registros y contraerla conforme se eliminan registros.

Al solicitar espacio de almacenamiento adicional, el usuario debe proporcionar a la función nueva una indicación de la cantidad de almacenamiento necesaria. Esto se hace solicitando suficiente espacio para un tipo de datos particular. Por ejemplo, la expresión `new(int)` o `new int` (las dos formas pueden usarse de manera intercambiable) solicita

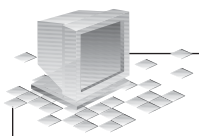
suficiente espacio de almacenamiento para almacenar un número entero. Una solicitud de almacenamiento suficiente para una estructura de datos se hace de la misma forma. Por ejemplo, usando la declaración

```
struct Tipo_tel
{
  string nombre;
  string num_telef;
};
```

las expresiones `newTipo_tel` y `new(Tipo_tel)` reservan suficiente almacenamiento para una estructura de datos `Tipo_tel`.

Al asignar almacenamiento en forma dinámica, no se tiene indicación anticipada del lugar en que el sistema de cómputo reservará físicamente el número de bytes solicitado, y no se tiene ningún nombre explícito para tener acceso a las ubicaciones de almacenamiento recién creadas. Para proporcionar acceso a estas ubicaciones, `new` devuelve la dirección de la primera ubicación que se ha reservado. Esta dirección debe ser asignada, por supuesto, a un apuntador. La devolución de un apuntador por `new` es útil en especial para crear una lista vinculada de estructuras de datos. Conforme se crea cada estructura nueva, el apuntador devuelto por `new` a la estructura puede asignarse a un miembro de la estructura previa en la lista.

El programa 13.10 ilustra el uso de `new` para crear una estructura en forma dinámica en respuesta a una solicitud de entrada del usuario.



Program 13.10

```
// a program illustrating dynamic structure allocation

#include <iostream>
#include <string>
using namespace std;

struct TeleType
{
  string name;
  string phoneNo;
};

void populate(TeleType *);  // function prototype needed by main(
void dispOne(TeleType *);   // function prototype needed by main(
```

(Continued)

```
int main()
{
  char key;
  TeleType *recPoint;   // recPoint is a pointer to a
                        // structure of type TeleType

  cout << "Do you wish to create a new record (respond with y or n): ";
  key = cin.get();
  if (key == 'y')
  {
    key = cin.get();      // get the Enter key in buffered input
    recPoint = new TeleType;
    populate(recPoint);
    dispOne(recPoint);

  }
  else
    cout << "\nNo record has been created.";

  return 0;
}

  // input a name and phone number
void populate(TeleType *record) // record is a pointer to a
  {                             // structure of type TeleType
    cout << "Enter a name: ";
    getline(cin,record->name);
    cout << "Enter the phone number: ";
    getline(cin,record->phoneNo);

    return;
  }
  // display the contents of one record
void dispOne(TeleType *contents)  // contents is a pointer to a
{                                 // structure of type TeleType
  cout << "\nThe contents of the record just created is:"
       << "\nName: " << contents->name
       << "\nPhone Number: " << contents->phoneNo << endl;

  return;
}
```

Una sesión de muestra producida por el programa 13.10 es

```
Desea crear un registro nuevo (responda con s o n): s
Introduzca un nombre: Muñoz, Jaime
Introduzca el número telefónico: (555) 617-1817

El contenido del registro que se acaba de crear es:
Nombre: Muñoz, Jaime
Número telefónico: (555) 617-1817
```

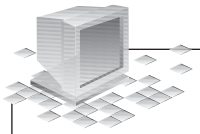
Al revisar el programa 13.10 se nota que sólo se hacen dos declaraciones de variables en `main()`. La variable clave se declara como una variable de carácter y la variable `regApunt` se declara como un apuntador a una estructura del tipo `Tipo_tel`. Debido a que la declaración para el tipo `Tipo_tel` es global, `Tipo_tel` puede usarse dentro de `main()` para definir `regApunt` como un apuntador a una estructura del tipo `Tipo_tel`.

Si un usuario introduce `s` en respuesta a la primera petición en `main()`, se hace una llamada a `new` por la memoria requerida para almacenar la estructura designada. Una vez que se ha cargado `regApunt` con la dirección apropiada, esta dirección puede usarse para tener acceso a la estructura recién creada. La función `poblar()` se usa para pedir al usuario los datos necesarios para llenar la estructura y almacenar los datos introducidos por el usuario en los miembros correctos de la estructura. El argumento transmitido a `poblar()` en `main()` es el apuntador `regApunt`. Como todos los argumentos transmitidos, el valor contenido en `regApunt` es transmitido a la función. Dado que el valor en `regApunt` es una dirección, `poblar()` recibe la dirección de la estructura recién creada y puede tener acceso directo a los miembros de la estructura.

Dentro de `poblar()`, el valor recibido se almacena en el argumento llamado `registro`. Dado que el valor que se almacenará en `registro` es la dirección de una estructura, `registro` debe declararse como un apuntador a una estructura. Esta declaración es proporcionada por la instrucción `Tipo_tel *registro;`. Las instrucciones dentro de `poblar()` usan la dirección en `registro` para localizar a los miembros respectivos de la estructura.

La función en `desplUno()` en el programa 13.10 se usa para desplegar el contenido de la estructura recién creada y poblada. La dirección transmitida a `desplUno()` es la misma dirección que se transmitió a `poblar()`. Debido a que este valor transmitido es la dirección de una estructura, el nombre del argumento usado para almacenar la dirección se declara como un apuntador al tipo de estructura correcto.

Una vez que se entiende el mecanismo de la llamada a `new`, puede usarse esta función para construir una lista de estructuras vinculadas. Como se describió en la sección anterior, las estructuras usadas en una lista vinculada deben contener al menos un miembro apuntador. La dirección en el miembro apuntador es la dirección inicial de la siguiente estructura en la lista. Además, debe reservarse un apuntador para la dirección de la primera estructura, y al miembro apuntador de la última estructura en la lista se le da una dirección `NULL` para indicar que no se apunta a más miembros. El programa 13.11 ilustra el uso de `new` para construir una lista vinculada de nombres y números telefónicos. La función `poblar()` usada en el programa 13.11 es la misma función usada en el programa 13.10, mientras que la función `desplegar()` es la misma función usada en el programa 13.9.



Program 13.11

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

const int MAXRECS = 3;   // maximum no. of records

struct TeleType
{
  char name;
  char phoneNo;
  TeleType *nextaddr;
};

void populate(TeleType *);   // function prototype needed by main(
void display(TeleType *);    // function prototype needed by main(

int main()
{
  int i;
  TeleType *list, *current; // two pointers to structures of
                            // type TeleType

    // get a pointer to the first structure in the list
  list = new TeleType;
  current = list;

    // populate the current structure and create the remaining str
  for(i = 0; i < MAXRECS - 1; i++)
  {
    populate(current);
    current->nextaddr = new TeleType;
    current = current->nextaddr;
  }

  populate(current);         // populate the last structure
  current->nextaddr = NULL;  // set the last address to a NULL add
  cout << "\nThe list consists of the following records:\n";
  display(list);   // display the structures

  return 0;
}
```

(Continued)

```
  // input a name and phone number
void populate(TeleType *record) // record is a pointer to a
{                               // structure of type TeleType

  cout << "Enter a name: ";
  getline(cin,record->name);
  cout << "Enter the phone number: ";
  getline(cin,record->phoneNo);

  return;
}

void display(TeleType *contents) // contents is a pointer to a
{                                      // structure of type TeleType
  while (contents != NULL)        // display till end of linked list
  {
    cout << endl << setiosflags(ios::left)
        << setw(30) << contents->name
        << setw(20) << contents->phoneNo;
    contents = contents->nextaddr;
  }
  cout << endl;

  return;
}
```

La primera vez que se llama a `new` en el programa 13.11 se usa para crear la primera estructura en la lista vinculada. Como tal, la dirección devuelta por `new` se almacena en la variable apuntadora llamada `lista`. La dirección en `lista` se asigna luego al apuntador llamado actual. Esta variable apuntadora siempre es usada por el programa para apuntar a la estructura actual. Debido a que la estructura actual es la primera estructura creada, la dirección en el apuntador llamado `lista` es asignada al apuntador llamado `actual`.

Dentro del ciclo `for` de `main()`, los miembros nombre y número telefónico de la estructura recién creada son poblados llamando a `poblar()` y transmitiendo la dirección de la estructura actual a la función. Con la devolución de `poblar()`, se asigna una dirección al miembro apuntador de la estructura `actual`. Esta dirección es la dirección de la siguiente estructura en la lista, la cual se obtiene de `new`. La llamada a `new` crea la siguiente estructura y devuelve su dirección al miembro apuntador de la estructura `actual`. Esto completa la población del miembro `actual`. La instrucción final en el ciclo `for` reinicia la dirección en el apuntador `actual` con la dirección de la siguiente estructura en la lista.

Después que se ha creado la última estructura, las instrucciones finales en `main()` pueblan esta estructura, asignando una dirección `NULL` al miembro apuntador, y llama a `desplegar()` para desplegar todas las estructuras en la lista. A continuación se proporciona una muestra de la ejecución del programa 13.11:

```
Introduzca un nombre: Amador, Samuel
Introduzca el número telefónico: (555) 898-2392
Introduzca un nombre: Domínguez, Edith
Introduzca el número telefónico: (555) 682-3104
Introduzca un nombre: López, Juan
Introduzca el número telefónico: (555) 718-4581

La lista consiste en los siguientes registros:

Amador, Samuel                (555) 898-2392
Domínguez, Edith              (555) 682-3104
López, Juan                   (555) 718-4581
```

Así como `new` crea almacenamiento en forma dinámica mientras se ejecuta un programa, la función `delete` regresa un bloque de almacenamiento a la computadora mientras el programa se ejecuta. El único argumento requerido por `delete` es la dirección inicial de un bloque de almacenamiento que fue asignado en forma dinámica. Por tanto, cualquier dirección devuelta por `new` puede transmitirse después a `delete` para regresar la memoria reservada a la computadora. La función `delete` no altera la dirección que se le transmite, sino tan sólo elimina el almacenamiento al que hace referencia la dirección.

Ejercicios 13.5

1. Como se describió en la tabla 13.1, el operador `new` devuelve la dirección de la primera área de almacenamiento nueva asignada o `NULL` si no se dispone de almacenamiento suficiente. Modifique el programa 13.11 para comprobar que se ha devuelto una dirección válida antes que se haga una llamada a `poblar()`. Despliegue un mensaje apropiado si no se dispone de almacenamiento suficiente.

2. Escriba una función en C++ llamada `eliminar()` que elimina una estructura existente de la lista de estructuras vinculadas creada por el programa 13.11. El algoritmo para eliminar una estructura vinculada deberá seguir la secuencia desarrollada para eliminar una estructura desarrollada en el ejercicio 4 en la sección 13.4. El argumento transmitido a `eliminar()` deberá ser la dirección de la estructura que precede al registro que se eliminará. En la función de eliminación, asegúrese que el valor del apuntador en la estructura eliminada reemplaza al valor del miembro apuntador de la estructura precedente antes que se elimine la estructura.

3. Escriba una función llamada `insertar()` que inserte una estructura en la lista de estructuras vinculadas creada en el programa 13.11. El algoritmo para insertar una estructura en una lista vinculada deberá seguir la secuencia para insertar un registro ilustrada antes en la figura 13.10. El argumento transmitido a `insertar()` deberá ser la dirección de la estructura que precede a la estructura que se va a insertar. La estructura insertada deberá seguir a esta estructura actual. La función `insertar()` deberá crear una estructura nueva en forma dinámica, llamar a la función `poblar` usada en el programa 13.11 y ajustar todos los valores apuntadores en forma apropiada.

4. Se desea insertar una estructura nueva en la lista de estructuras vinculadas creada por el programa 13.11. La función desarrollada para hacer esto en el ejercicio 3 suponía que se conocía la dirección de la estructura precedente. Escriba una función llamada `hallarReg()` que devuelva la dirección de la estructura inmediatamente precedente al punto en que se va a insertar la estructura nueva. (*Sugerencia:* `hallarReg()` debe solicitar el nuevo nombre como entrada y comparar el nombre introducido con los nombres existentes para determinar dónde colocar el nuevo nombre.)

5. Escriba una función en C++ llamada `modificar()` que pueda utilizarse para modificar los miembros nombre y número telefónico de una estructura del tipo creado en el programa 13.11. El argumento transmitido a `modificar()` deberá ser la dirección de la estructura que se va a modificar. La función `modificar()` deberá desplegar primero el nombre y número telefónico existentes en la estructura seleccionada y luego solicitar datos nuevos para estos miembros.

6. **a.** Escriba un programa en C++ que presente inicialmente un menú de opciones para el usuario. El menú deberá consistir en las siguientes opciones:

   Crear una lista vinculada inicial de nombres y números telefónicos
   Insertar una estructura nueva en la lista vinculada
   Modificar una estructura existente en la lista vinculada
   Eliminar de la lista una estructura existente
   Salir del programa

   Con la selección del usuario, el programa deberá ejecutar las funciones apropiadas para satisfacer la solicitud.

   b. ¿Por qué la creación original de una lista vinculada por lo general es realizada por un programa y las opciones para agregar, modificar o eliminar una estructura en la lista es proporcionada por un programa diferente?

13.6 UNIONES[2]

Una unión es un tipo de datos que reserva la misma área en la memoria para dos o más variables, cada una de las cuales puede ser un tipo de datos diferente. Una variable que se declara como un tipo de datos de unión puede usarse para contener una variable de carácter, una variable en número entero, una variable de precisión doble o cualquier otro tipo de datos válido en C++. Cada uno de estos tipos, pero sólo uno a la vez, puede asignarse en realidad a la variable unión.

La definición de una unión tiene la misma forma que una definición de estructura, con la palabra clave `union` usada en lugar de la palabra clave `struct`. Por ejemplo, la declaración

```
union
{
  char clave;
  int num;
  double voltios;
} val;
```

[2]Este tema puede omitirse en la primera lectura sin perder la continuidad temática.

crea una variable de tipo unión llamada `val.` Si `val` fuera una estructura consistiría en tres miembros individuales. Sin embargo, como una unión, `val` contiene un solo miembro que puede ser una variable de carácter llamada `clave`, una variable en número entero llamada `num` o una variable en precisión doble llamada `voltios`. En efecto, una unión reserva suficientes ubicaciones de memoria para acomodar su tipo de datos del miembro más grande. Este mismo conjunto de ubicaciones es referenciado luego por diferentes nombres de variables, dependiendo del tipo de datos del valor que reside en la actualidad en las ubicaciones reservadas. Cada valor almacenado sobrescribe el valor anterior, usando tantos bytes del área de memoria reservada como sea necesario.

Los miembros individuales de la unión se referencian usando la misma notación que para los miembros de la estructura. Por ejemplo, si la unión `val` se está usando en la actualidad para almacenar un carácter, el nombre correcto de la variable para tener acceso al carácter almacenado es `val.clave`. Del mismo modo, si la unión se usa para almacenar un número entero, se tiene acceso al valor con el nombre `val.num` y se tiene acceso a un valor de precisión doble con el nombre `val.voltios`. Al usar miembros de unión, es responsabilidad del programador asegurar que se use el nombre de miembro correcto para el tipo de datos que reside en la actualidad en la unión.

Por lo general se usa una segunda variable para seguirle la pista al tipo de datos actual almacenado en la unión. Por ejemplo, podría usarse el siguiente código para seleccionar el miembro apropiado de `val` para desplegarlo. Aquí el valor en la variable `tipo_u` determina el tipo de datos almacenado en la actualidad en la unión `val`.

```
switch (tipo_u)
{
  case ‘c’: cout << val.clave;
            break;
  case ‘d’: cout << val.voltios;
            break;
  case 'd': cout << val.volts;
            break;
  default : cout << "Tipo inválido en tipo_u : " << tipo_u;
}
```

Como sucede con las estructuras, un tipo de datos puede asociarse con una unión. Por ejemplo, la declaración

```
union Fecha_tiempo
{
  int dias;
  double tiempo;

};
```

proporciona un tipo de datos unión sin reservar en realidad ninguna ubicación de almacenamiento. Este tipo de datos puede usarse entonces para definir cualquier número de variables. Por ejemplo, la definición

```
Fecha_tiempo primero, segundo, *pt;
```

crea una variable de unión llamada `primero`, una variable de unión llamada `segundo` y un apuntador que puede usarse para almacenar la dirección de cualquier unión que tenga la forma `Fecha_tiempo`. Una vez que se ha declarado un apuntador a una unión, la mis-

ma notación usada para tener acceso a los miembros de una estructura puede usarse para tener acceso a los miembros de la unión. Por ejemplo, si se hace la asignación `pt = &primero;`, entonces `pt->fecha` hace referencia al miembro `fecha` de la unión llamada primero.

Las uniones en sí pueden ser miembros de estructuras o arreglos. Estructuras, arreglos y apuntadores también pueden ser miembros de uniones. En cada caso, la notación usada para tener acceso a un miembro debe ser consistente con la anidación empleada. Por ejemplo, en la estructura definida por

```
struct
{
  char tipo_u;
  union
  {
    char *texto;
    float tasa;
  } impuesto_u;
}  bandera;
```

se hace referencia a la variable tasa como

```
bandera.impuesto_u.tasa
```

Del mismo modo, se hace referencia al primer carácter de la cadena cuya dirección está almacenada en el apuntador `texto` como

```
bandera.impuesto_u.texto
```

Ejercicios 13.6

1. Suponga que se ha hecho la siguiente definición

   ```
   union
   {
     double tasa;
     double impuestos;
     int num;
   } bandera;
   ```

   Para esta unión escriba activaciones de flujo `cout` apropiadas para desplegar los diversos miembros de la unión.

2. Defina una variable de unión llamada `auto` que contenga un número entero llamado `anio`, un arreglo de 10 caracteres llamado `nombre` y un arreglo de 10 caracteres llamado `modelo`.

3. Defina una variable de unión llamada `lang` que permitiría que se haga referencia a un número de precisión doble con los nombres de variables `voltios` y `fem`.

4. Declare un tipo de datos de unión llamado `cant` que contenga una variable en número entero llamada `cant_int`, una variable en precisión doble llamada `cant_doble` y un apuntador a un carácter llamado `ptClave`.

5. a. ¿Qué piensa que se desplegará con la siguiente sección de código?

```
union
{
  char caracter;
  double tipob;
} alt;
alt.caracter = 's';
cout << alt.tipob;
```

b. Incluya el código presentado en el ejercicio 5a en un programa y ejecute el programa para verificar su respuesta al ejercicio 5a.

13.7 ERRORES COMUNES DE PROGRAMACIÓN

Con frecuencia se cometen tres errores comunes cuando se usan estructuras o uniones. El primer error ocurre debido a que las estructuras y uniones, como entidades completas, no pueden usarse en expresiones relacionales. Por ejemplo, aun si `Tipo_tel` y `Tipo_aparato` son dos estructuras del mismo tipo, la expresión `Tipo_tel == Tipo_aparato` es inválida. Por supuesto, los miembros individuales de una estructura o unión pueden compararse si son del mismo tipo de datos, usando cualquiera de los operadores relacionales de C++.

El segundo error común en realidad es una extensión de un error de apuntador en su relación con las estructuras y uniones. Siempre que se usa un apuntador para "apuntar a" cualquiera de estos tipos de datos, o siempre que un apuntador es en sí mismo un miembro de una estructura o una unión, tenga cuidado de usar la dirección en el apuntador para tener acceso al tipo de datos apropiado. Si no se tiene claro a qué está apuntando, recuerde: "Si tiene dudas, imprímalo".

El error final se relaciona de manera específica con las uniones. Dado que una unión puede almacenar sólo uno de sus miembros a la vez, debe tener cuidado de seguirle la pista a la variable almacenada en la actualidad. Almacenar un tipo de datos en una unión y tener acceso a él con el nombre de variable equivocado puede producir un error que es particularmente problemático de localizar.

13.8 RESUMEN DEL CAPÍTULO

1. Una estructura permite que se agrupen variables individuales bajo un nombre de variable común. Se tiene acceso a cada variable en una estructura por su nombre de variable de estructura, seguido por un punto, seguido por su nombre de variable individual. Otro término para una estructura de datos es registro. Una forma para declarar una estructura es:

```
struct
{
  declaraciones de miembros individuales;
} nombreEstructura;
```

2. Puede crearse un tipo de datos a partir de una estructura usando la forma de declaración

```
struct TipoDatos
{
  declaraciones de miembros individuales;
};
```

   Las variables de estructura individuales pueden definirse entonces como este `TipoDatos`. Por convención, la primera letra del nombre `TipoDatos` siempre se escribe con mayúsculas.

3. Las estructuras son útiles en particular como elementos de arreglos. Usadas de esta manera, cada estructura se convierte en un registro en una lista de registros.

4. Pueden usarse estructuras completas como argumentos de función, en cuyo caso la función llamada recibe una copia de cada elemento en la estructura. También puede transmitirse la dirección de una estructura, ya sea como una referencia o como un apuntador, lo cual le proporciona a la función llamada acceso directo a la estructura.

5. Los miembros de la estructura puede ser cualquier tipo de datos válido en C++, incluyendo otras estructuras, uniones, arreglos y apuntadores. Cuando se incluye un apuntador como un miembro de la estructura, puede crearse una lista vinculada. Dicha lista usa el apuntador en una estructura para "apuntar a" (contener la dirección de) la siguiente estructura lógica en la lista.

6. Las uniones se declaran de la misma manera que las estructuras. La definición de una unión crea un área de memoria superpuesta, con cada miembro de la unión usando las mismas ubicaciones de almacenamiento de memoria. Por tanto, sólo un miembro de una unión puede estar activo a la vez.

Parte cuatro

Temas adicionales

Capítulos

14 Métodos numéricos

15 Operaciones bit por bit

CAPÍTULO 14
Métodos numéricos

TEMAS

14.1 INTRODUCCIÓN AL CÁLCULO DE RAÍCES

14.2 EL MÉTODO DE BISECCIÓN

14.3 REFINAMIENTOS AL MÉTODO DE BISECCIÓN
- EL MÉTODO REGULA FALSI
- EL MÉTODO REGULA FALSI MODIFICADO
- RESUMEN DE LOS ALGORITMOS BASADOS EN LA BISECCIÓN

14.4 EL MÉTODO DE LA SECANTE

14.5 INTRODUCCIÓN A LA INTEGRACIÓN NUMÉRICA

14.6 LA REGLA TRAPEZOIDAL
- FORMA DE CÁLCULO DE LA ECUACIÓN PARA LA REGLA TRAPEZOIDAL
- EJEMPLO DE UN CÁLCULO DE REGLA TRAPEZOIDAL

14.7 LA REGLA DE SIMPSON
- EJEMPLO DE LA REGLA DE SIMPSON COMO UNA APROXIMACIÓN A UNA INTEGRAL

14.8 ERRORES COMUNES DE PROGRAMACIÓN

14.9 RESUMEN DEL CAPÍTULO

*Una de las tareas más comunes en las ciencias y la ingeniería es encontrar las raíces de ecuaciones; es decir, dada una función f(*x*), encontrar valores de* x *tales que f(*x*) = 0.0. Este tipo de problema también incluye determinar los puntos de intersección de dos curvas. Si las curvas están representadas por funciones* f(x) *y* g(x)*, respectivamente, los puntos de intersección corresponden a las raíces de la función* F(x) = f(x) – g(x).

Una segunda tarea importante es la integración numérica, en la cual se utilizan métodos de aproximación para determinar el valor de la integral cuando no existen soluciones exactas. En este capítulo se presentan y se aplican primero varias técnicas de programación para encontrar las raíces de ecuaciones. Esto es seguido por las técnicas más comunes empleadas en la integración numérica.

14.1 INTRODUCCIÓN AL CÁLCULO DE RAÍCES

Las técnicas para encontrar raíces son importantes por diversas razones. Son útiles, fáciles de entender y por lo general son fáciles de llevar a cabo. Por tanto, con un mínimo de instrucción podrá solucionar problemas genuinos en ingeniería. Son elementos vitales en el análisis numérico apreciar lo que se puede resolver o no y comprender con claridad la precisión de las respuestas obtenidas. Dado que esta apreciación y comprensión llegan sobre todo con la experiencia, es necesario comenzar a resolver problemas numéricos de inmediato. Además, descubrirá que resolver problemas para obtener raíces es divertido.

Algunos ejemplos de los tipos de funciones que se encuentran en los problemas de obtención de raíces son

$$ax^2 + bx + c = 0 \tag{14.1}$$

$$2x^4 - 7x^3 + 4x^2 + 7x - 6 = (x - 2)(x - 1)(x + 1)(2x - 3) = 0 \tag{14.2}$$

$$x^5 - 2x^3 - 5x^2 + 2 = 0 \tag{14.3}$$

$$\text{sen}^5(x) + \text{sen}^3(x) + 5\cos(x) - 7 = 0 \tag{14.4}$$

$$100e - x - \text{sen}(2\pi x) = 0 \tag{14.5}$$

La ecuación cuadrática general, ecuación 14.1, puede ser resuelta con facilidad y exactitud usando la fórmula cuadrática

$$r = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$$

La ecuación 14.2 puede ser resuelta exactamente para x al factorizar el polinomio. Las raíces entonces son 1, −1, 2, 3/2. Sin embargo, la mayor parte de los polinomios no puede factorizarse con tanta facilidad, y se requieren otras técnicas más generales. Hay fórmulas para la solución exacta de ecuaciones cúbicas o cuadráticas generales, pero son muy complicadas y por tanto rara vez se usan. No es posible usar una fórmula exacta para un polinomio como la ecuación 14.3, en la cual la potencia más elevada de x es mayor que 4. Para determinar las raíces en estos polinomios por lo general deberán utilizarse medios numéricos.

Como se recordará de los cursos de álgebra en el bachillerato, un polinomio de grado n (es decir, la potencia más elevada de x^n) tiene precisamente n raíces, de las cuales algunas pueden ser números complejos y otras pueden ser raíces múltiples. Por tanto, la ecuación 14.3 tiene tres raíces reales,

$$r_1 = -0.712780744625\ldots$$
$$r_2 = 0.57909844162\ldots$$
$$r_3 = 2.0508836199\ldots$$

y dos raíces complejas,

$$r_4 = 0.757225433526 + i(0.57803468208)$$
$$r_5 = 0.757225433526 - i(0.57803468208)$$

La ecuación

$$x^2 - 2x + 1 = 0$$

puede factorizarse como

$$(x - 1)^2 = 0$$

y tiene dos raíces reales, las cuales resultan ser iguales. En este caso se dice que la raíz tiene una raíz múltiple con multiplicidad 2.

Las ecuaciones 14.4 y 14.5 se llaman **ecuaciones trascendentales** y representan una clase de funciones diferente por completo. Las ecuaciones trascendentales por lo general implican funciones trigonométricas, exponenciales o logarítmicas y no pueden reducirse a ninguna ecuación polinomial en x. Las raíces reales de los polinomios por lo general se clasifican como números racionales (es decir, una fracción simple) o irracionales (por ejemplo, $\sqrt{2}$). Las raíces de las ecuaciones trascendentales con frecuencia son números trascendentales como π o e. Los números irracionales y los números trascendentales se representan con fracciones decimales que no se repiten y no pueden expresarse como fracciones simples. Estos números son importantes para las matemáticas, ya que a ellos se debe que el sistema de números reales sea denso o continuo. Por tanto, la clasificación de ecuaciones como polinomios o trascendentales y las raíces de estas ecuaciones como racionales o irracionales es vital para la matemática tradicional; sin embargo, la distinción tiene menos consecuencias para la computadora. De hecho, el sistema numérico con el que trabaje la computadora no sólo no es continuo, es un conjunto finito.

De todos modos, cuando se obtienen las raíces de las ecuaciones, es innecesaria la distinción entre ecuaciones polinomiales y trascendentales y se aplican los mismos procedimientos numéricos a ambos. Sin embargo, la distinción entre los dos tipos de funciones es importante en otros aspectos. Muchos de los teoremas que aprendió relacionados con las raíces de los polinomios no se aplican a las ecuaciones trascendentales. Por ejemplo, las ecuaciones 14.4 y 14.5 tienen un número infinito de raíces reales.

Todas las técnicas para obtener raíces expuestas en este capítulo son interactivas; es decir, se especifica un intervalo que se sabe que contiene una raíz o tan sólo una suposición inicial para la raíz y las diversas rutinas devolverán un intervalo más limitado o una suposición mejor. Algunos de los esquemas que se expondrán garantizan que con el tiempo se encontrará una raíz pero se puede requerir de un tiempo de computadora considerable para llegar a la respuesta. Otros pueden converger en una raíz mucho más rápido pero son más susceptibles a presentar problemas de divergencia; es decir, no están garantizados.

El ingrediente común en todas las recetas para obtener raíces es que es mejor evitar las dificultades potenciales de cálculo de cualquier naturaleza al dominar con la mayor inteligencia posible la elección inicial del método usado y la suposición inicial respectiva. Esta parte del problema con frecuencia es la más difícil y la que lleva más tiempo. El arte involucrado en el análisis numérico consiste en equilibrar el tiempo para optimizar la solución del problema antes del cálculo, frente al tiempo que se pasa corrigiendo los errores imprevistos durante el cálculo. Si es posible, la función deberá ser esbozada antes de intentar la obtención de raíces, ya sea usando rutinas de graficación o generando una tabla de valores de función que luego se grafica en forma manual. Estas gráficas son útiles en extremo para el programador no sólo para estimar la primera suposición para la raíz, sino también para anticipar dificultades potenciales. Si no es factible hacer un esbozo, se debe utilizar algún método para verificar la función y comprender lo que ésta realiza antes de iniciar el cálculo real. Como ejemplo del procedimiento general a seguir considérese la función trascendental

$$f(x) = e^{-x} - \text{sen}(\tfrac{1}{2}\pi x)$$

Esta ecuación, como se verá, tiene un número infinito de raíces positivas. Por el momento, nos concentraremos en obtener una suposición inicial para la primera raíz, para luego utilizar esta suposición inicial para encontrar un valor más preciso de la raíz.

Comenzaremos por recopilar tanta información como sea posible antes de tratar de construir un programa en C++. Este paso casi siempre implica hacer un esbozo de la función. La ecuación previa puede escribirse como

$$e^{-x} = \text{sen}(\tfrac{1}{2}\pi x)$$

Una raíz de esta ecuación corresponde entonces a cualquier valor de x tal que el lado izquierdo y el lado derecho sean iguales. Si los lados izquierdo y derecho se grafican de manera independiente, las raíces de la ecuación original están dadas entonces por los puntos de intersección de las dos curvas (véase la figura 14.1). A partir del esbozo se puede observar que las raíces son

Raíces ≈ 0.4, 1.9, 4.0, . . .

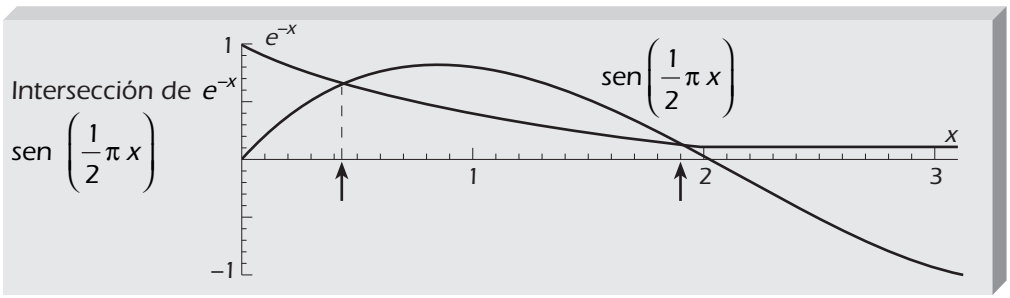


Figura 14.1 *Gráfica de e^{-x} y sen(½πx) para localizar los puntos de intersección.*

Dado que el seno oscila, habrá un número infinito de raíces positivas. Nos concentraremos primero en mejorar la estimación de la primera raíz cerca de 0.4. Se inicia por establecer un procedimiento, o algoritmo, que se basa en el método más obvio de ataque cuando se usa una calculadora de bolsillo; es decir, se comienza con algún valor de x justo antes de la raíz (como 0.3) y se avanza a través del eje x, observando con cuidado la magnitud y en particular el signo de la función.

Paso	x	e^{-x}	sen(½πx)	$f(x) = e^{-x} - \text{sen}(\frac{1}{2}\pi x)$
0	0.3	0.741	0.454	0.297
1	0.4	0.670	0.588	0.082
2	0.5	0.606	0.707	−0.101

Hay que observar que la función ha cambiado de signo entre 0.4 y 0.5, lo que indica una raíz entre estos dos valores de x. Por tanto, para la siguiente aproximación se utilizará el valor intermedio, $x = 0.45$, para el siguiente paso:

Paso	x	e^{-x}	sen(½πx)	$f(x) = e^{-x} - \text{sen}(½\pi x)$
3	0.45	0.638	0.649	−0.012

La función es negativa de nuevo en 0.45, lo que indica que la raíz está entre 0.4 y 0.45. La siguiente aproximación será, por consiguiente, el punto intermedio de este intervalo, 0.425. De esta manera se puede proceder de manera sistemática al cálculo de la raíz con cualquier grado deseado de precisión.

Paso	x	e^{-x}	sen(½πx)	$f(x) = e^{-x} - \text{sen}(½\pi x)$
4	0.425	0.654	0.619	0.0347
5	0.4375	0.6456	0.6344	0.01126
6	0.44365	0.6417	0.6418	−0.00014

El elemento clave en el procedimiento es vigilar el signo de la función. Cuando cambia el signo, se emprende una acción específica para depurar la estimación de la raíz. Este cambio en el signo de la función, que indica que se ha localizado la proximidad de una raíz, forma el elemento clave en el código de computadora para localizar raíces. En las siguientes tres secciones se consideran varios métodos para la obtención de raíces que utiliza este procedimiento como base.

Ejercicios 14.1

1. Use la técnica iterativa presentada en esta sección para encontrar una raíz de la ecuación

$$f(x) = \text{sen } x - x/3 = 0 \quad (x \text{ está en radianes})$$

Para hacer esto, primero vuelva a escribir la ecuación como

$$\text{sen } x = x/3$$

y grafique los lados izquierdo y derecho de manera independiente en la misma gráfica.

14.2 El método de bisección

El procedimiento para obtener raíces ilustrado en la sección anterior es adecuado para cálculos manuales; sin embargo, una ligera modificación lo hará más "sistemático" y más fácil de adaptar al código de computadora.

Supóngase que ya se sabe que hay una raíz entre $x = a$ y $x = b$; es decir, la función cambia de signo en este intervalo. Por simplicidad se asumirá que sólo hay una raíz entre $x = a$ y $x = b$ y que la función es continua en este intervalo.

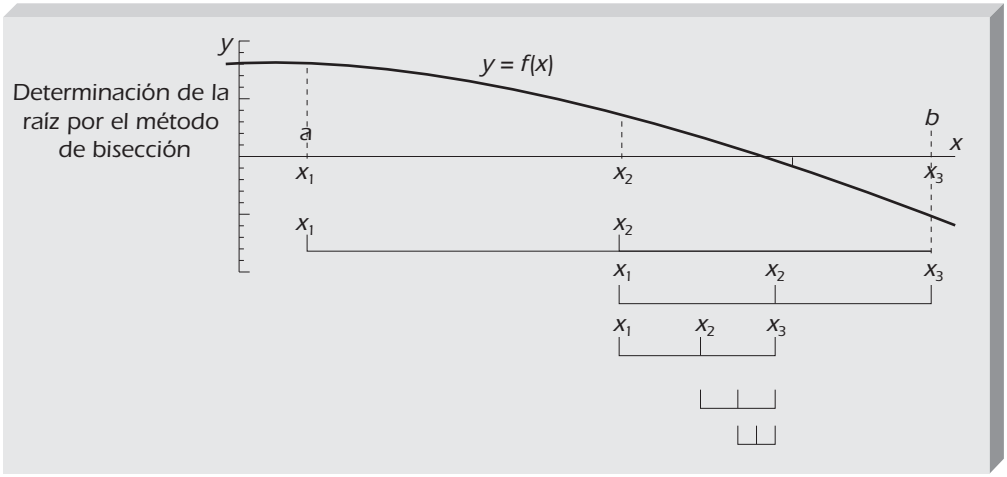


Figura 14.2 Un esbozo de una función con una raíz entre *a* y *b*.

La función podría parecerse entonces al esbozo de la figura 14.2. Si a continuación se definen $x_1 = a$ y $x_3 = b$ como los extremos izquierdo y derecho del intervalo, respectivamente, y $x_2 = \frac{1}{2}(x_1 + x_3)$ como el punto intermedio, ¿en cuál mitad del intervalo cruza la función el eje *x*? En la figura, el cruce se encuentra a la derecha, así que se reemplaza el intervalo completo por la mitad derecha del intervalo. Por tanto,

x_2 ahora se vuelve x_1

x_3 permanece como está

x_2 se vuelve a calcular como el valor $\frac{1}{2}(x_1 + x_3)$

y se plantea de nuevo la pregunta "¿en cuál mitad del intervalo cruza la función el eje *x*?" Después de determinar por segunda vez si la mitad izquierda o la mitad derecha contiene la raíz, el intervalo es reemplazado una vez más por la mitad izquierda o derecha del intervalo. Este proceso se continúa hasta que la raíz se encuentre dentro de algún rango de precisión asignado con anterioridad. Cada paso reduce a la mitad el intervalo, y por tanto después de *n* iteraciones, el tamaño del intervalo que contiene la raíz será $(b - a)/2^n$. Si se requiere encontrar una raíz dentro de una tolerancia δ; es decir, $|x - \text{raíz}| < \delta$, el número de iteraciones *n* requeridas puede determinarse de

$$\frac{b - a}{2^n} < \delta$$

Por ejemplo, el intervalo de búsqueda inicial en el ejemplo presentado en la sección anterior fue $(b - a) = 0.1$. Si se requiere que la raíz tenga una precisión de $\delta = 10^{-5}$, entonces

$$\frac{0.1}{2^n} < 10^{-5}$$

o

$$2^n > 10^4$$

Esto puede solucionarse para n como sigue:

$$n > \frac{\log(10^4)}{\log(2)} > 13$$

Por tanto, el cálculo revela que el grado de precisión deseado se logra después de la decimotercera aplicación del procedimiento de reducir a la mitad el intervalo. El único elemento del método que se ha omitido es la forma en que la computadora determina cuál mitad del intervalo contiene el cruce del eje. Para ese fin, considérese el producto de la función evaluada a la izquierda, $f_1 = f(x_1)$, y la función evaluada en el punto intermedio, $f_2 = f(x_2)$.

Sí	Entonces
$f_1 f_2 > 0.0$	f_1 y f_2 son ambos positivos o negativos. En cualquier caso, no hay cruce entre x_1 y x_2, y no se encuentra ninguna raíz en el intervalo.
$f_1 f_2 < 0.0$	$f(x)$ ha cambiado de signo entre x_1 y x_2. En este caso, hay una raíz dentro del intervalo delimitado por x_1 y x_2.

Un programa para calcular las raíces de una ecuación usando este procedimiento se ilustra en el programa 14.1. Al revisar este programa, deberán observarse en especial las siguientes características:

- En cada iteración después de la primera, sólo hay una evaluación de la función. Sería muy ineficiente reevaluar $f(x_1)$, $f(x_2)$ y $f(x_3)$ para cada iteración dado que dos de ellas ya se conocen. Si la función fuera complicada en extremo, cálculos redundantes como éste serían un problema grave. Puede desperdiciarse una gran cantidad de tiempo de computadora con evaluaciones de función innecesarias.
- El programa contiene varias comprobaciones para problemas potenciales junto con mensajes de diagnóstico (por ejemplo, iteración excesiva, ninguna raíz en el intervalo, etc.), aun cuando el programador pueda pensar que estas posibilidades son remotas. En general, entre más verificaciones de éstas contiene un programa es mejor. Sólo requieren de unos cuantos minutos para codificarse y pueden ahorrar horas de depuración.
- El criterio para el éxito se basa en el tamaño del intervalo. Por tanto, aun si la función no estuviera cerca de cero en un punto, x cambia muy poco, y continuar no mejoraría de manera considerable la precisión de la raíz.

Programa 14.1

```
#include <iostream>
#include <cmath>
using namespace std;

void biseccion(double, double, double, int); // prototipo de función
double f(double); // prototipo de función
using namespace std;

int main()
{
  int imax;          // número máximo de iteraciones
  double a, b;       // extremos izquierdo y derecho del intervalo original
  double epsilon;    // criterio de convergencia

  // obtiene los datos de entrada
  cout << "Introduzca los límites del intervalo de búsqueda original, a y b: ";
  cin  >> a >> b;
  cout << "Introduzca los criterios de convergencia: ";
  cin  >> epsilon;
  cout << "Introduzca el numero maximo de iteraciones permitidas: ";
  cin  >> imax;

  biseccion(a, b, epsilon, imax);

  return 0;
}

// Una función de bisección que encuentra raices de una función
// Se sabe que el intervalo a < x < b contiene una raiz de f(x). La estimación
// de la raiz se mejora sucesivamente al hallar en cual mitad del intervalo
// se encuentra la raiz y luego reemplazando el intervalo original por esa
   mitad del intervalo.
//
void biseccion(double a, double b, double epsilon, int imax)
{
  int i;             // contador de iteraciones actuales
  double x1, x2, x3; // izquierda, derecha y punto intermedio del intervalo
                        actual
  double f1, f2, f3; // función evaluada en estos puntos
  double width;      // ancho del intervalo original = (b - a)
  double curwidth;   // ancho del intervalo actual = (x3 - x1)

  // repetición de los datos de entrada transmitidos
  cout << "\nEl intervalo de busqueda original es de " << a << " a "
       << b << endl;
  cout << "El criterio de convergencia es: intervalo < " << epsilon << endl;
  cout << "El número máximo de iteraciones permitido es " << imax << endl;

  // calcula la raiz
```

(Continúa)

(Continuación)

```
    x1 = a;
    x3 = b;
    f1 = f(x1);
    f3 = f(x3);
    ancho = (b - a);

    // verifica si hay una raiz en el intervalo
    if (f1 * f3 > 0.0)
      cout << "\nNo existe una raiz en el intervalo original" << endl;
    else
    {
      for (i = 1; i <= imax; i++)
      {
        // encuentra cual mitad del intervalo contiene la raiz
        x2 = (x1 + x3) / 2.0;
        f2 = f(x2);
        if (f1 * f2 <= 0.0) // la raiz esta en la mitad izquierda del intervalo
        {
          ancho_actual = (x2 - x1) / 2.0;
          f3 = f2;
          x3 = x2;
        }
        else // la raiz esta en la mitad derecha del intervalo
        {
          ancho_actual = (x3 - x2) / 2.0;
          f1 = f2;
          x1 = x2;
        }
        if (ancho_actual < epsilon)
        {
          cout << "\nSe encontró una raíz en x = " << x2 << was found
               << "en " << i << " iteraciones" << endl;
          cout << "El valor de la función es " << f2 << endl;
          return;
        }
      }
    }
    cout << "\nDespués de " << imax << " iteraciones, no se encontró ninguna raiz "
         << "dentro del criterio de convergencia" << endl;

    return;
}

// funcion para evaluar f(x)
double f(double x)
{
    const double PI = 2*asin(1.0); // valor de pi

    return (exp(-x) - sen(0.5 * PI * x));
}
```

Una ejecución de muestra usando el programa 14.1 produjo lo siguiente:

```
Introduzca los límites del intervalo de búsqueda original, a y b: .4 .5
Introduzca los criterios de convergencia: .00001
Introduzca el número máximo de iteraciones permitidas: 25

El intervalo de búsqueda original es de 0.4 a 0.5
El criterio de convergencia es: intervalo < 1e-005
El número máximo de iteraciones permitido es 25

Se encontró una raíz en x = 0.443567 en 13 iteraciones
El valor de la función es 1.22595e-005
```

Aunque el programa 14.1 se usa para evaluar las raíces de la ecuación $f(x) = e^x - \text{sen}(\pi x/2)$, al cambiar el cálculo en la instrucción de devolución en la última función, `f()`, enlistada en el programa, este programa puede utilizarse para cualquier función.

Un comentario final: el método de bisección usado en el programa 14.1 es un ejemplo de un método llamado de fuerza bruta; es decir, posee un mínimo de delicadeza. Aunque ilustra las técnicas fundamentales de C++, se dispone de procedimientos numéricos más poderosos e ingeniosos. Estas técnicas se presentan en la siguiente sección.

Ejercicios 14.2

1. Use el procedimiento de bisección para encontrar las raíces de una ecuación. Primero elabore una tabla de la siguiente forma:

Paso	x_1	x_2	x_3	$f(x_1)$	$f(x_2)$	$f(x_3)$	Punto de cruce	
							Izquierda	Derecha
0	a	$\frac{1}{2}(a+b)$	b					
1								
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.

A continuación, use el procedimiento de bisección y una calculadora de bolsillo para obtener las raíces de las siguientes funciones con una precisión de cinco cifras significativas.

a. $f(x) = x^2 + 2x - 15$ (Use $a = 2.8$, $b = 3.1$. La respuesta exacta = 3.0)
b. $g(x) = \frac{1}{2}\text{sin}(x)(e^x - e^{-x})$ (Ésta es la ecuación de engranaje elíptico, con x en radianes. Use $a = 1$, $b = 4$)
c. $E(x) = \sqrt{(R^2 - x^2)} - x\tan(x)$ (Ésta es la ecuación para energías cuánticas de una partícula en una caja. Use $R = 10$, $a = 4.0$, $b = 4.7$)
d. Prediga el número de pasos necesarios para obtener la respuesta a la precisión especificada de cinco cifras significativas en los incisos del a al c.

2. a. Modifique el programa 14.1 para resolver las raíces indicadas de cada una de las funciones del ejercicio 1.

b. Modifique el programa 14.1 para producir una tabla similar a la requerida en el ejercicio 1 para cada una de las funciones enlistadas en el ejercicio 1.

3. Escriba un programa en C++ para encontrar el máximo de una función $f(x)$ en un intervalo $a \le x \le b$ empezando en $x = a$ con un tamaño del paso Δx. Evalúe $f_1 = f(x)$ y $f_2 = f(x + \Delta x)$. Si $f^1 < f^2$, reemplace x con $x + \Delta x$ y continúe; de lo contrario, reduzca el tamaño del paso a la mitad y repita la comparación. El programa deberá terminar con éxito cuando $\Delta < 10^{-6}$.

14.3 REFINAMIENTOS AL MÉTODO DE BISECCIÓN

El método de bisección descrito en la sección anterior presenta los fundamentos sobre los cuales están construidos la mayor parte de los métodos para obtener raíces. Es un método llamado de fuerza bruta que rara vez se usa en la práctica, dado que casi para cualquier problema se dispone de un método alternativo que es más rápido, más preciso y sólo ligeramente más complejo. Todos los refinamientos del método de bisección que podrían diseñarse se basan en intentos de usar tanta información como esté disponible acerca del comportamiento o la función en cada iteración. En el método de bisección ordinario, la única característica de la función que se vigila es su signo. Por tanto, si se buscaran raíces de la función

$$f(x) = e^{-x} - \text{sen}(\tfrac{1}{2}\pi x)$$

se comenzaría la búsqueda, como se describe en la sección 14.1, recorriendo el eje x y observando un cambio en el signo de la función, como sigue.

i	x_i	$f(x_i)$
0	0.0	2.0
1	0.1	1.33
2	0.2	0.75
3	0.3	0.29
4	0.4	–.05

El paso siguiente en el procedimiento de bisección es reducir el tamaño del paso a la mitad; es decir, intentar $x_5 = 0.35$. Sin embargo, por la magnitud de los números anteriores se esperaría que la raíz estuviera más cerca de 0.4 que de 0.3. Por tanto, tomando en cuenta la información sobre el tamaño del valor funcional además de su signo, puede acelerarse la convergencia. En el presente caso podría interpolarse la raíz para que sea aproximadamente

$$\frac{0.29 - 0.0}{0.29 - (-0.05)} = \frac{f_3 - 0}{f_3 - f_4} = 0.853$$

de la distancia de $x_3 = 0.3$ a $x_4 = 0.4$, o $x_5 = 0.3853$. Continuando de esta manera e interpolando cada paso, se obtendrían los siguientes resultados:

i	x_i	$f(x_i)$
3	0.30	0.29
4	0.40	–0.05
5	0.385	–0.0083
6	0.3823	–0.0013
7	0.3819	–0.00019
8	0.38185	–0.000028
9	0.38184	–0.000004

Al comparar estos resultados con el método de bisección aplicado a una función similar en la sección anterior, se puede observar que la velocidad de la convergencia para el presente método es significativamente más rápida. La siguiente tarea es formalizar este procedimiento en un método adecuado para una función general.

El método regula falsi

La idea básica en el primer refinamiento del algoritmo de bisección es que el nuevo método será en esencia el mismo que la bisección con excepción que en lugar de usar el punto intermedio del intervalo en cada paso del cálculo, se usa un valor interpolado para la raíz. El método se ilustra en la figura 14.3. En ella se sabe que existe una raíz en el intervalo $(x_1 \leftrightarrow x_3)$, y en el dibujo f_1 es negativo, mientras f_3 es positivo. La posición interpolada de la raíz es x_2. Dado que los dos triángulos, ABC y CDE, son semejantes, las longitudes de los lados están relacionados por

$$\frac{\text{DE}}{\text{AB}} = \frac{\text{CD}}{\text{BC}}$$

o

$$\frac{0.0 - f_1}{f_3 - f_1} = \frac{x_2 - x_1}{x_3 - x_1}$$

lo cual puede resolverse para la posición desconocida x_2 para producir

$$x_2 = x_1 - (x_3 - x_1)\frac{f_1}{f_3 - f_1}$$

Este valor de x_2 reemplaza entonces el punto intermedio utilizado en el algoritmo de bisección, y el resto del procedimiento permanece exactamente igual. Por tanto, el siguiente paso sería determinar si la raíz real está a la izquierda o a la derecha de x_2. Como antes,

Si $f_1 \times f_2 < 0$ entonces la raíz está a la izquierda

Si $f_2 \times f_3 < 0$ entonces la raíz está a la derecha

En la figura la raíz está a la izquierda de x_2, así que el intervalo usado para la siguiente iteración sería

$$x_3 = x_2$$
$$f_3 = f_2$$
$$x_2 = x_1 - (x_3 - x_1)\frac{f_1}{f_3 - f_1}$$
$$f_2 = f(x_2)$$

En otras palabras, para emplear este algoritmo ligeramente más rápido, el único cambio que tiene que hacerse al código de bisección previo es reemplazar las instrucciones de la forma

$$x_2 = (x_1 + x_3) / 2.$$

por una instrucción de la forma

$$x_2 = x_1 - (x_3 - x_1)\frac{f_1}{f_3 - f_1}$$

Este método todavía garantiza que finalmente se obtendrá una raíz y casi siempre convergirá más rápido que el algoritmo de bisección convencional. Sin embargo, se paga un precio pequeño. Los valores de f_1 y f_3 usados para solucionar x_2 pueden ser casi iguales, y podrían plagarnos de errores de redondeo en su diferencia. Además, en el algoritmo de bisección podría predecirse con alguna precisión el número de iteraciones requeridas para obtener la raíz con una precisión deseada (véase la sección 14.1). Esta producción ya no es posible si se usan los valores interpolados, y el código debe incluir ahora una comprobación de iteraciones excesivas.

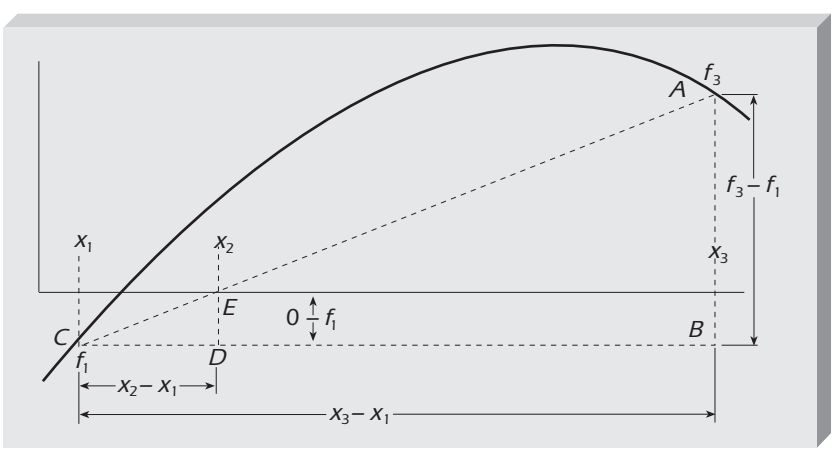


Figura 14.3 Estimación de la raíz por interpolación.

Este método ilustra que un cambio casi trivial en el algoritmo, el cual se basa en una vigilancia más inteligente de la función, puede cosechar recompensas considerables en una convergencia más rápida. El nombre formal del método que se acaba de describir es **método regula falsi** (el método de la posición falsa).

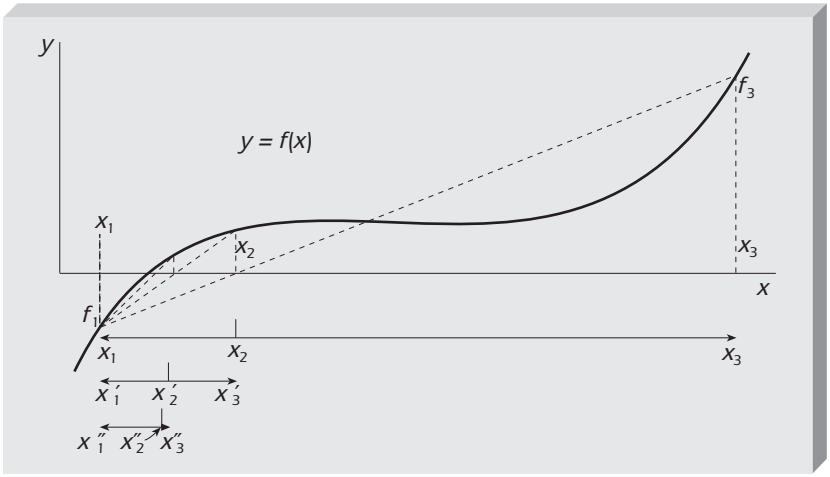


Figura 14.4 *Ilustración gráfica de varias iteraciones del algoritmo regula falsi.*

¿Hay algunas mejoras adicionales en el algoritmo básico de bisección que puedan implementarse con facilidad? Para responder esta cuestión se debe examinar con más detalle la manera en que el método *regula falsi* llega a una solución. Esto se hace mejor de manera gráfica. El cálculo iniciado en la figura 14.3 se continúa en la figura 14.4. Hay que observar que en este ejemplo, en el cual la función es cóncava hacia abajo cerca de la raíz, el valor del límite izquierdo del intervalo de búsqueda cerca de la raíz, x_1, nunca cambia. La raíz real siempre permanece en el segmento izquierdo en cada iteración. El segmento derecho del intervalo, $x_3 - x_2$, se reduce con bastante rapidez; pero no es así con el segmento izquierdo, $x_2 - x_1$. Si la función fuera cóncava hacia arriba, ocurriría lo contrario. Por tanto, un inconveniente en el método *regula falsi* es que aun cuando el método converge más rápido a un valor de x que produce un $|f(x)|$ "pequeño", el intervalo que contiene la raíz no disminuye de manera significativa.

El método regula falsi modificado

Quizás el procedimiento puede hacerse convergir más rápido si de alguna manera puede lograrse que el intervalo se colapse desde ambas direcciones. Una forma de lograr esto se demuestra en la figura 14.5. La idea es como sigue:

Si se determina que la raíz se encuentra en el segmento izquierdo $(x_2 - x_1)$
  La línea de interpolación se traza entre los puntos $(x_1, \frac{1}{2}f_1)$ ***y*** (x_3, f_3)
De no ser así, si la raíz está en el segmento derecho
  La línea de interpolación se traza entre los puntos (x_1, f_1) ***y*** $(x_3, \frac{1}{2}f_3)$
Termina el si

Usando este algoritmo, la pendiente de la línea se reduce *artificialmente*. El efecto de esta reducción es que si la raíz está en el lado izquierdo del intervalo original, con el tiempo aparecerá en el segmento derecho de un intervalo posterior y en lo subsiguiente alternará entre el izquierdo y el derecho. La última modificación al método de bisección en combinación con el método *regula falsi* se conoce como el *método regula falsi modificado*, un procedimiento muy potente y popular para encontrar las raíces de ecuaciones. Las alter-

nativas al código de bisección original presentado en el programa 14.1 son bastante triviales y se incorporan dentro del programa 14.2.

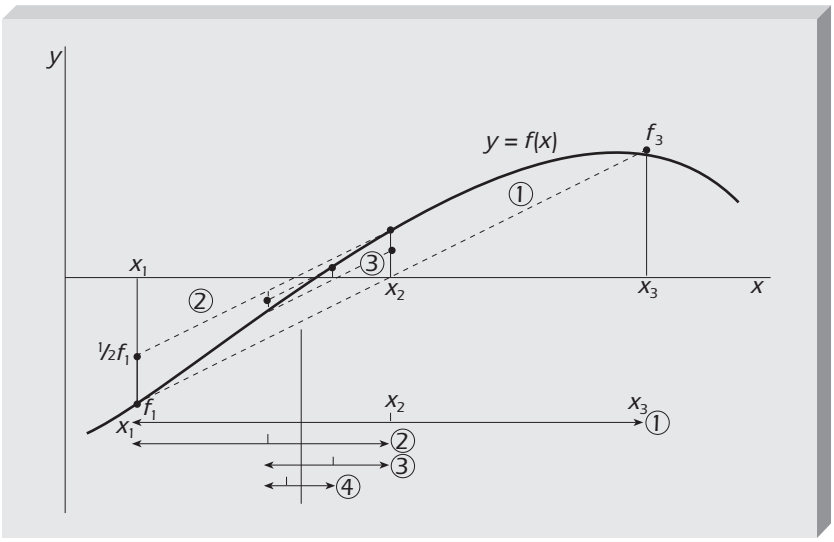


Figura 14.5 *Ilustración gráfica del método regula falsi modificado.*

El código de C++ en el programa 14.2 requiere alguna explicación:

1. La función $f(x)$ se evalúa sólo una vez por ciclo. Si la función es complicada y por consiguiente su cálculo es costoso, esta medida de eficiencia puede ser atractiva, incluso decisiva, al elegir el método de solución apropiado.
2. El código sólo puede terminar de tres maneras.
   a. Una ruta de éxito: si el tamaño fraccional actual del intervalo de búsqueda [es decir (intervalo actual)/(intervalo original)] es menor que el criterio de convergencia suministrado por el usuario. De ser así, se ha logrado el objetivo original del programa de agrupar una raíz por un escaso margen. No hay garantía de que este criterio producirá un valor de $f(x)$ que sea "pequeño". Sin embargo, el punto es que iteraciones sucesivas sólo han producido cambios pequeños en el intervalo que contiene la raíz y por tanto continuar el proceso no es necesario ni productivo.
   b. Hay dos rutas de error: 1) si el número de iteraciones es mayor que $I_{máx}$, el número máximo de iteraciones establecido por el programador, entonces se detiene el proceso. Esta prueba permite al programador especificar el costo máximo que se aceptará para un intento de solución. Dado que en los métodos *regula falsi* y *regula falsi* modificado el número de iteraciones no es predecible, es esencial esta forma de salvaguarda. También es una precaución prudente contra errores imprevistos en la construcción del problema que pudieran causar que el programa entrara en un ciclo infinito y no obtenga una solución. Instrucciones de este tipo se requieren en cualquier programa en el que exista el peligro de entrar en un ciclo infinito. 2) Si la función no cambia de signo ($f_1 \times f_3 > 0$), entonces el proceso se detiene. Dado que se sabe que el

intervalo original contiene una raíz, la única manera en que puede surgir esta condición es por error. Por lo general el error está en el código para la función $f(x)$; es decir, se está intentando encontrar una raíz de una función diferente de la que se pretende.

Programa 14.2

```
#include <iostream>
#include <cmath>
using namespace std;

void regfalsimod(double, double, double, int); // prototipo de función
double f(double); // prototipo de función

int main()
{
  int imax;           // número máximo de iteraciones
  double a, b;        // extremos izquierdo y derecho del intervalo original
  double epsilon;     // criterio de convergencia

  // obtiene los datos de entrada
  cout << "Introduzca los límites del intervalo de búsqueda original, a y b: ";
  cin  >> a >> b;
  cout << "Introduzca los criterios de convergencia: ";
  cin  >> epsilon;
  cout << "Introduzca el número máximo de iteraciones permitidas: ";
  cin  >> imax;

  regfalsimod(a, b, epsilon, imax);

  return 0;
}

// Una función regula falsi modificada que encuentra raices de una función
// El número máximo de iteraciones permitidas es imax. El criterio
// de convergencia es el tamanho fraccional del intervalo de busqueda (x3 - x1) / (b - a)
// es menor que epsilon. Se usa un factor de relajacion RELAX
void regfalsimod(double a, double b, double epsilon, int imax)
{
  const double RELAX = 0.9; // el factor de relajacion

  int i;             // contador de iteracion actual
  double x1, x2, x3; // izquierda, derecha y punto intermedio de intervalo actual
  double f1, f2, f3; // función evaluada en estos puntos
  double width;      // ancho del intervalo original = (b - a)
  double curwidth;   // ancho del intervalo actual = (x3 - x1)
```

(Continúa)

(Continuación)

```
// repetición de los datos de entrada transmitidos
cout << "\nEl intervalo de búsqueda original es de " << a << " a " << b
     << "\nEl criterio de convergencia es: intervalo < " << epsilon
     << "\nEl número máximo de iteraciones permitidas es " << imax << endl;

// calcula la raiz
x1 = a;
x3 = b;
f1 = f(x1);
f3 = f(x3);
ancho = abs(b - a);

// iteraciones
for (i = 1; i <= imax; i++)
{
  ancho_actual = (x3 - x1) / ancho;
  x2 = x1 - ancho * ancho_actual * f1 / (f3 - f1);
  f2 = f(x2);
  if (abs(ancho_actual) < epsilon) // se encuentra la raiz
  {
    cout << "\nSe encontró una raiz en x = " << x2 << endl
         << "en " << i << " iteraciones" << endl;
    cout << "El valor de la función es " << f2 << endl;
    return;
  }
  else // comprueba el cruce a izquierda y derecha
  {
    if(f1 * f2 < 0.0) // comprueba el cruce a la izquierda
    {
      x3 = x2;
      f3 = f2;
      f1 = RELAX * f1;
    }
    else if (f2 * f3 < 0.0) // comprueba el cruce a la derecha
    {
      x1 = x2;
      f1 = f2;
      f3 = RELAX * f3;
    }
    else // no hay cruce en el intervalo
    {
      cout << "La búsqueda de una raiz ha fallado debido a que no hay raiz en
               el intervalo\n"
           << "En el paso " << i << " de la iteración la función no cambia de signo"
        << endl;
    }
  }
}
cout << "\nDespués de " << imax << " iteraciones, no se encontró ninguna raiz "
     << "dentro del criterio de convergencia\n"
     << "La búsqueda de una raiz ha fallado debido a excesivas iteraciones\n"
     << "después del número máximo de " << imax << " iteraciones" << endl;
```

(Continúa)

(Continuación)

```
  return;
}
// funcion para evaluar f(x)
double f(double x)
{
  const double PI = 2*asin(1.0); // valor de pi

  return (exp(-x) - sen(0.5 * PI * x));
}
```

Una muestra de la ejecución del programa 14.2 produjo lo siguiente:

```
Introduzca los límites del intervalo de búsqueda original, a y
b: .4 .5
Introduzca los criterios de convergencia: .00001
Introduzca el número máximo de iteraciones permitidas: 25

El intervalo de búsqueda original es de 0.4 a 0.5
El criterio de convergencia es: intervalo < 1e-005
El número máximo de iteraciones permitidas es 25

Se encontró una raiz en x = 0.443574 en 7 iteraciones
El valor de la función es -2.25374e-009
```

Al comparar los resultados de la muestra de la ejecución del programa 14.2 con los del programa 14.1, se puede observar que se localizó una raíz más exacta con la función *regula falsi* modificada con seis iteraciones menos (7 frente a 13). La exactitud de la raíz está indicada por el valor funcional en la raíz, el cual está más cerca de 0.0 en la salida producida por el programa 14.2. En la tabla 14.1 se presenta una comparación más completa de la velocidad de convergencia para los tres métodos para encontrar raíces presentados hasta ahora, según se aplican estos métodos a la función

$$f(x) = 2e^{-2x} - \text{sen}(\pi x)$$

se muestra en la tabla 14.1.

Se usó un factor de reducción de la pendiente de una mitad al elaborar el dibujo en la figura 14.5 y es un ejemplo de lo que se conoce como **factor de relajación**, un número usado para alterar los resultados de una iteración antes de insertarlos en la siguiente. Determinar el factor de relajación óptimo casi siempre es un problema complejo en extremo en cualquier cálculo y rebasa el alcance de este texto. Sin embargo, en este caso un poco de ensayo y error muestra que una disminución menos drástica en la pendiente producirá una convergencia mejorada. Usar un factor de relajación de 0.9 deberá ser adecuado para la mayor parte de los problemas; este factor se usó para generar los valores en la tabla 14.1.

Tabla 14.1 Comparación de los métodos para encontrar raíces usando la función $f(x) = 2e^{-2x} - \text{sen}(\pi x)$

i	Bisección x_2	Bisección regula x_2	Bisección regula modificada x_2
1	0.35	0.385	0.385
2	0.375	0.3823	0.3820
3	0.3875	0.3819	0.38183
4	0.38125	0.38185	0.381843
5	0.38438	0.381844	0.38184267
6	0.38281	0.381843	0.38184276
7	0.38203	0.3818428	0.38184275
8	0.38164	0.38184275	0.38184275

Resumen de los algoritmos basados en la bisección

Los aspectos característicos de los tres métodos expuestos en esta sección y la anterior se enlistan a continuación:

Bisección	Éxito basado en el tamaño del intervalo Convergencia lenta Número predecible de iteraciones Intervalo reducido a la mitad en cada iteración Se garantiza que se agrupa una raíz
Regula falsi	Éxito basado en el tamaño de la función Convergencia más rápida Número de iteraciones impredecible El intervalo que contiene la raíz no es pequeño Vigila el tamaño de la función al igual que su signo
Regula falsi modificado	Éxito basado en el tamaño del intervalo Convergencia más rápida Número de iteraciones impredecible

De los tres métodos, es probable que el *regula falsi* modificado sea el más eficiente para problemas comunes y es el algoritmo recomendado siempre que la única información disponible es que la función cambia de signo entre x_1 y x_3.

El requerimiento de que el intervalo de búsqueda inicial sea aquel en el que la función cambie de signo (sólo una vez) puede ser problemático en ocasiones. Por ejemplo, el problema de encontrar la raíz de la función

$$f(x) = x^2 - 2x + 1 = (x - 1)^2$$

no es adecuado para ninguno de los algoritmos basados en la bisección dado que la función nunca cambia de signo. Esta dificultad ocurre siempre que la raíz de la función es una raíz múltiple de multiplicidad par. Un método que supera esta limitación es el método de la secante, el cual es el tema de la siguiente sección.

Ejercicios 14.3

1. Reproduzca aproximadamente el esbozo de la figura 14.6 y luego aplique en forma gráfica el método *regula falsi* en tres iteraciones.

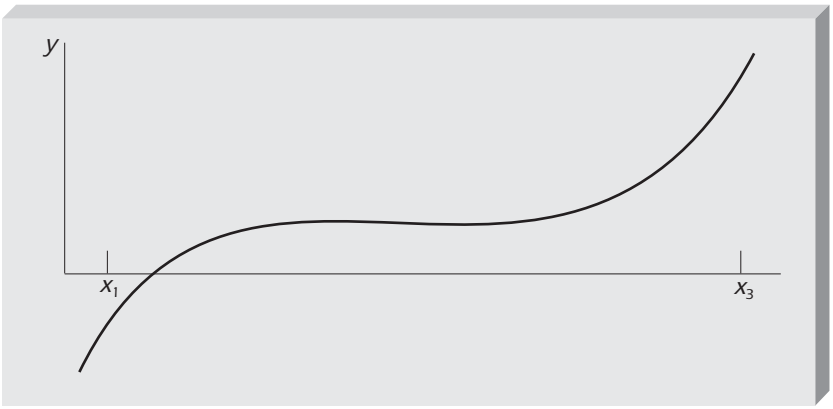


Figura 14.6 Función para el ejercicio 1.

2. Usando una calculadora de bolsillo, aplique el procedimiento *regula falsi* en tres iteraciones para las siguientes funciones:

a. $f(x) = xe^{-x^2} - \cos(x)$; $a = 0$, b = 2; raíz exacta = 1.351491185. . .
b. $g(x) = x^2 - 2x - 3$; $a = 0$, $b = 4$; raíz exacta = 3.0
c. $h(x) = e^x - (1 + x + x^2/2)$; $a = -1$, $b = 1$; raíz exacta = 0.0
d. $F(x) = x^3 - 2x - 5$; $a = 1$, $b = 3$; raíz exacta = 2.0945514815. . .
e. $G(x) = 10 \ln(x) - x$; $a = 1$, $b = 2$; raíz exacta = 1.1183255916. . .

3. Reproduzca aproximadamente la figura 14.6 y luego aplique en forma gráfica el método regula falsi modificado en tres iteraciones.

4. **a.** Ejecute el programa 14.2.
b. Use el programa 14.2 para encontrar la raíz de una de las funciones en el ejercicio 2 con una precisión de 10^{-5}.
c. Cambie el factor de relajación de 0.9 a 0.75 y vuelva a ejecutar el programa. Comente la diferencia entre los dos cálculos.

14.4 EL MÉTODO DE LA SECANTE

El método de la secante es idéntico al método de la posición falsa (método *regula falsi*) excepto que el signo de $f(x)$ no necesita comprobarse en cada iteración. Como en el método *regula falsi*, se requieren los valores de x_0 y x_1 para comenzar el procedimiento pero luego se emplea el siguiente algoritmo para obtener una mejora para el siguiente valor de x:

Empezar con el intervalo definido por (x_0,x_1)
Calcular el siguiente valor de x como

$$x_2 = \frac{f(x_0)}{f(x_0) - f(x_1)} \left(x_1 - x_0\right)$$

Reemplazar el par de valores (x_0,x_1) con el par (x_1,x_2)
Repetir los pasos 2 y 3 hasta que el valor de $f(x)$ esté dentro de un límite de cero aceptable

Puede mostrarse que el método de la secante (véase el ejercicio 2) es equivalente a reemplazar en forma repetida la función por líneas rectas trazadas a través de los puntos $[x_0,f(x_0)]$ y $[x_1,f(x_1)]$; es decir, líneas secantes. En el programa 14.3 se incluye el código en C++ para que una función implemente el método de la secante.

Una muestra de ejecución usando el programa 14.3 produjo lo siguiente:

```
Introduzca los límites del intervalo de búsqueda original, a y
b: .4 .5
Introduzca los criterios de convergencia: .00001
Introduzca el número máximo de iteraciones permitidas: 25

El intervalo de búsqueda original es de 0.4 a 0.5
El criterio de convergencia es: intervalo < 1e-005
El número máximo de iteraciones permitidas es 25

Se encontró una raiz en x = 0.443567 en 4 iteraciones
El valor de la función es -9.014174e-008
```

Aunque es probable que el método de la secante sea el método más popular para encontrar la raíz de una función, plantea problemas de divergencia. Por ello, para este método deberá incorporarse una comprobación en el código en C++ para detectar cuando los intervalos sucesivos comienzan a ser más grandes en lugar de volverse más pequeños.

Programa 14.3

```
#include <iostream>
#include <cmath>
using namespace std;

void secante(double, double, double, int); // prototipo de función
double f(double); // prototipo de función

int main()
{
  int imax;          // número máximo de iteraciones
  double a, b;       // extremos izquierdo y derecho del intervalo original
  double epsilon;    // criterio de convergencia

  // obtiene los datos de entrada
  cout << "Introduzca los límites del intervalo de búsqueda original, a y b: ";
  cin  >> a >> b;
  cout << "Introduzca los criterios de convergencia: ";
  cin  >> epsilon;
  cout << "Introduzca el número máximo de iteraciones permitidas: ";
  cin  >> imax;

  secante(a, b, epsilon, imax);

  return 0;
}

// Esta función implementa el método de la secante para encontrar
// una raiz de una funcion
void secante(double a, double b, double epsilon, int imax)
{

  int i;              // contador de iteraciones actuales
  double x0, x1;     // valores izquierdo y derecho de x del intervalo actual
  double f0, f1;      // función evaluada en estos puntos
  double dx0;         // delta x0
  double dx1;         // delta x1

  // repeticion de los datos de entrada transmitidos
  cout << "\nEl intervalo de búsqueda original es de " << a << " a " << b
       << "El criterio de convergencia es: intervalo < " << epsilon
       << "El número máximo de iteraciones permitido es " << imax << endl;
```

(Continúa)

(Continuación)

```
    // determina la raiz
    x0 = a;
    f0 = f(x0);
    dx0 = abs(b - a);

    // iteraciones
    for (i = 1; i <= imax; i++)
    {
      x1 = x0 + dx0;
      f1 = f(x1);
      if (abs(f1) < epsilon) // se encuentra la raiz
      {
        cout << "\nSe encontró una raiz en x = " << x1 + dx1 << endl
             << "en " << i << " iteraciones" << endl;
        cout << "El valor de la funcion es " << f1 << endl;
        return;
      }
      else // hace la siguiente iteracion
      {
        dx1 = ( f1/(f0 - f1) ) * dx0;
        x0 = x1;
        dx0 = dx1;
        f0 = f1;
      }
    }
    cout << "\nDespués de " << imax << " iteraciones, no se encontro ninguna raiz "
         << "dentro del criterio de convergencia\n"
         << "La búsqueda de una raiz ha fallado debido a iteraciones excesivas\n"
         << "después del número máximo de " << imax << " iteraciones" << endl;

    return;
  }

  // funcion para evaluar f(x)
  double f(double x)
  {
    const double PI = 2*asin(1.0); // valor de pi

    return (exp(-x) - sen(0.5 * PI * x));
  }
```

Ejercicios 14.4

1. **a.** Ejecute el programa 14.3 en su computadora.
   b. Use el programa 14.3 para encontrar la raíz de la función $f(x) = x^2 - 2x + 0.9$ empezando con una suposición inicial de $x_0 = 0.6$ y $x_1 = 0.9$.

2. La función `secante` del programa 14.3 se usó para encontrar una raíz de la función $f(x) = x^2 - 2x + 0.9$ empezando con una suposición inicial de $x_0 = 0.6$, $x_1 = 0.9$ y se imprimieron los valores de x_0 y $x_1 - x_0$ para cada iteración. Los resultados son

Paso	x_0	$x_1 - x_0$
0	0.600	0.300
1	0.900	−0.180
2	0.720	−0.057
3	0.663	0.022
4	0.685	−0.001

   Grafique con cuidado la función para $0.5 \le x \le 1.0$ y use los números anteriores para demostrar de manera gráfica cómo llega el método de la secante a la raíz de la función.

3. Use el programa 14.3 para encontrar una raíz de la función $f(x) = x^2 - 2x - 3$.

14.5 INTRODUCCIÓN A LA INTEGRACIÓN NUMÉRICA

La integración de una función de una sola variable puede considerarse ya sea como lo opuesto de la diferenciación, es decir, la antiderivada, o como el área bajo una curva. Las antiderivadas por lo general se exponen a profundidad en un curso de cálculo. Aquí nos concentraremos en cambio en el enfoque menos analítico y más visual para interpretar una integral definida como un área. Es decir, la integral de la función $f(x)$ desde $x = a$ hasta $x = b$, designada como

$$I = \int_a^b f(x)\,dx$$

se evaluará ideando esquemas para medir el área bajo la gráfica de la función sobre su intervalo (véase la figura 14.7). Este método para evaluar una integral se presta de manera tan natural al cálculo numérico que la forma más efectiva para comprender el proceso de integración es aprender primero el enfoque numérico y luego reforzar estas ideas con los conceptos más formales de la antiderivada.

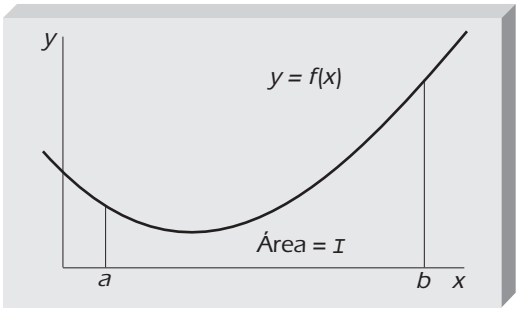


Figura 14.7 *Una integral como un área bajo una curva.*

Otra razón para estudiar la integración numérica en esta etapa es que es un presunto proceso estable; casi siempre funciona. Esto se debe a que la integración numérica consiste en expresar el área como la suma de áreas de segmentos más pequeños, un procedimiento que está relativamente libre de problemas como la división entre cero o el error de redondeo causados al sustraer números de aproximadamente la misma magnitud.

Por último, por desgracia es cierto que muchas, quizá la mayor parte de las integrales que se dan en los problemas de ingeniería o ciencias reales no pueden expresarse en ninguna forma cerrada.

Para integrar una función de manera formal; es decir, obtener una expresión cerrada como respuesta, con frecuencia se requieren una capacitación y experiencia considerables. Deben aprenderse y entenderse docenas de "trucos". Por otra parte, los procedimientos de integración numérica son menos en número, todos bastante fáciles de entender y recordar. Como en muchos procedimientos numéricos anteriores, se comienza por reemplazar la función en un rango limitado por segmentos de línea recta. El intervalo desde $x = a$ hasta $x = b$ se divide en subintervalos o paneles de tamaño Δx, la función es reemplazada por segmentos de línea en cada subintervalo, y el área bajo la función es aproximada entonces por el área bajo los segmentos de línea. Ésta es la regla trapezoidal para la aproximación de una integral y se describe en la siguiente sección. El siguiente orden de aproximación es reemplazar la función por segmentos parabólicos y se conoce como regla de Simpson, la cual se presenta en la sección 14.7.

14.6 LA REGLA TRAPEZOIDAL

Una aproximación al área bajo una curva complicada se obtiene al suponer que la función puede reemplazarse con funciones más simples en un rango limitado. Una línea recta, la aproximación más simple a una función, es la primera que se considera y conduce a lo que se conoce como **regla trapezoidal**.

El área bajo la curva $f(x)$ desde $x = a$ hasta $x = b$ es aproximada por el área debajo de una línea recta trazada entre los puntos x_a,$f(a)$ y x_b,$f(b)$ (véase la figura 14.8). El área más clara es entonces la aproximación a la integral y es el área de un trapezoide, el cual es

$I \simeq$ (valor promedio de f en el intervalo) (ancho del intervalo)

o

$$I \simeq \tfrac{1}{2}\,[f(a) + f(b)]\,(b - a) = T_0 \qquad (14.6)$$

Ésta es la regla trapezoidal para un panel, identificado como T_0.

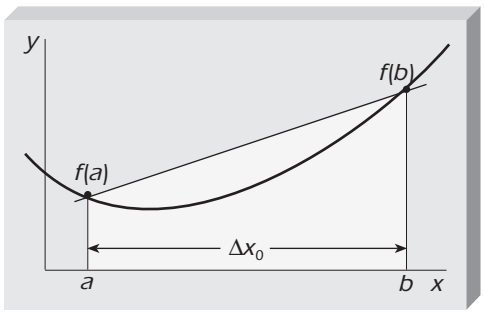


Figura 14.8 *Aproximación del área bajo una curva con un solo trapezoide.*

Para mejorar la precisión de la aproximación al área bajo una curva, a continuación el intervalo se divide a la mitad y la función es aproximada por segmentos de línea recta en cada mitad. El área en este caso es aproximada por el área de dos trapezoides, como se ilustra en la figura 14.9.

$$I \approx T_1 = \left[\tfrac{1}{2}\left(f(a) + f_1\right)\Delta x_L\right] + \left[\tfrac{1}{2}\left(f_1 + f(b)\right)\Delta x_L\right]$$

o

$$T_1 = \frac{\Delta x_L}{2}\left[f(a) + 2f_1 + f(b)\right] \qquad (14.7)$$

donde

$$\Delta x_L = \frac{(b-a)}{2}$$

$$f_1 = f\left(x = a + \Delta x_L\right)$$

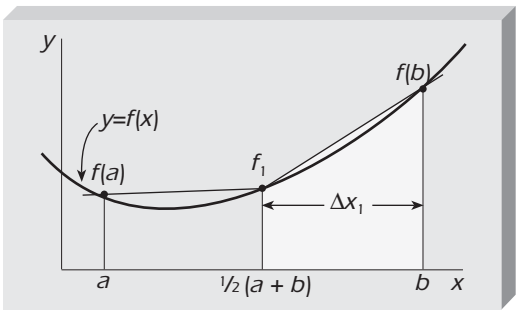


Figura 14.9 *Aproximación al área con dos paneles.*

Hay que observar que cuando se suman las áreas de los trapezoides, los lados en $f(a)$ y $f(b)$ son lados sólo del primer y último trapezoides, mientras el lado en f_1 es un lado de dos trapezoides y por tanto "cuenta doble", lo que explica el factor de 2 en la ecuación 14.7.

Además, la aproximación con dos paneles, T_1, puede relacionarse con los resultados de un panel, T_0, como

$$T_1 = \frac{T_0}{2} + \Delta x_1 f_1 \quad (14.8)$$

Para incrementar más la precisión, el intervalo simplemente se subdivide en un número mayor de paneles. Es obvio que el resultado para n paneles es

$$I \approx T_n = \tfrac{1}{2}\Delta x_n \left[f(a) + 2\sum_{i=1}^{n-1} f_i + f(b) \right] \quad (14.9)$$

donde $\Delta x_n = (b - a)/n$ y f_i es la función evaluada en cada uno de los puntos interiores,

$$f_i = f(x = a + i\,\Delta x_n)$$

La razón para el factor extra de 2 en la ecuación 14.9 es la misma que en el ejemplo de los dos paneles. La ecuación 14.9 se conoce como **regla trapezoidal.**

Forma de cálculo de la ecuación para la regla trapezoidal

La ecuación 14.9 se derivó suponiendo que todos los paneles tienen el mismo ancho y que éste es igual a Δx_n. Sin embargo, en la derivación no se requiere que los paneles tengan un ancho igual y la ecuación puede generalizarse con facilidad a una partición del intervalo en paneles desiguales de ancho Δx_i, $i = 1, \ldots, n - 1$. Sin embargo, por razones que se explicarán un poco más adelante, no sólo se restringirán los anchos de los paneles para que sean iguales, sino que el número de paneles debe ser una potencia de 2; es decir,

$$n = 2^k$$

El número de paneles es n, el orden del cálculo se llamará k y la correspondiente aproximación por la regla trapezoidal se denominará T_k. Por tanto, T_0 es el resultado para el panel $n = 2^0 = 1$. La situación para $k = 2$ o $2^2 = 4$ paneles se ilustra en la figura 14.10. En la

figura el ancho de un panel es $\Delta x_2 = (b - a)/2^2$ y el valor de la aproximación de $k = 2$ por la regla trapezoidal es

$$T_2 = (\Delta x_2/2)\left[f(a) + 2f(a + \Delta x_2) + 2f(a + \Delta x_2) + 2f(a + 3\Delta x_2) + f(b)\right] \quad (14.10)$$

Sin embargo, en vista que $2\Delta x_2 = \Delta x_1$, se puede observar que

$$f(a + 2\Delta x_2) = f(a + x_1)$$

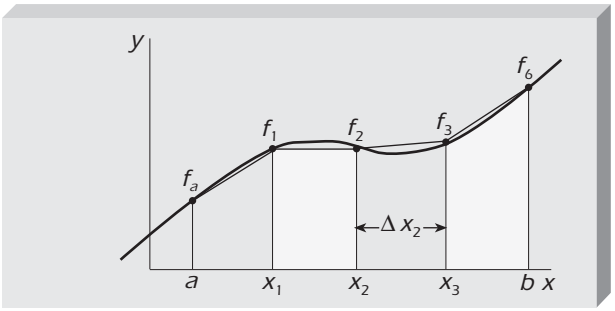


Figura 14.10 Aproximación trapezoidal de cuatro paneles, T_2.

y $f(a + \Delta x_1)$ ya se había determinado en el cálculo previo de T_1 (ecuación 14.8). El punto es que al duplicar sucesivamente el número de paneles en cada etapa, la única información nueva que se requiere para proceder al siguiente orden de la aproximación por la regla trapezoidal es la evaluación de la función en los puntos intermedios de los intervalos actuales.

Para explotar más este hecho, pueden utilizarse la ecuaciones 14.7 y 14.8 para replantear la ecuación 14.10 en la forma

$$\begin{aligned} T_2 &= (\Delta x_1/4)\left[f(a) + 2f(a + \Delta x_1) + f(b)\right] + \Delta x_2\left[f(a + \Delta x_2) + f(a + 3\Delta x_2)\right] \\ &= T_1/2 + \Delta x_2\left[f(a + \Delta x_2) + f(a + 3\Delta x_2)\right] \end{aligned}$$

Esta ecuación puede generalizarse con facilidad para producir

$$T_k = \tfrac{1}{2}T_{k-1} + \Delta x_k \sum_{\substack{i=1 \\ \text{sólo nones}}}^{n-1} f(a + i\Delta x_k) \quad (14.11)$$

donde

$$\Delta x_k = \frac{b - a}{2^k}$$

Entonces, el procedimiento para usar la ecuación 14.11 para aproximar una integral por la regla trapezoidal es:

1. Calcular T_0 usando la ecuación 14.6.
2. Aplicar de manera repetida la ecuación 14.11 para k = 1,2, ... hasta que se obtenga suficiente precisión.

Ejemplo de un cálculo de regla trapezoidal

Para ilustrar las ideas de esta sección, se usa la integral

$$I = \int_1^2 (1/x)\, dx$$

Por supuesto, la función $f(x) = 1/x$ puede integrarse analíticamente para dar $\ln(x)$, y dado que $\ln(1) = 0$, el valor de la integral es $\ln(2) = 0.69314718$. La aproximación de la regla trapezoidal a la integral con $a = 1$ y $b = 2$ comienza con la ecuación 14.6 para obtener T_0.

$$T_0 = \frac{1}{2}\left(\frac{1}{1} + \frac{1}{2}\right)(2 - 1) = 0.75$$

Entonces el uso repetido de la ecuación 14.11 produce

$k = 1$ $\quad \Delta x_1 = ½$

$$T_1 = T_0/2 + ½\left[f\left(1 + ½\right)\right] = 0.75/2 + ½\,(1/1.5)$$
$$= 0.708333$$

$k = 2$ $\quad \Delta x_2 = ¼$

$$T_2 = T_1/2 + ¼\,(1/1.25 + 1/1.75)$$
$$= 0.6970238$$

$k = 3$ $\quad \Delta x_3 = ⅛$

$$T_3 = T_2/2 + ⅛\,(1/1.25 + 1/1.375 + 1/1.625 + 1/1.875)$$
$$= 0.69412185$$

Continuar el cálculo hasta $k = 5$ produce

k	T_k
0	0.75
1	0.70833
2	0.69702
3	0.69412
4	0.69339
5	0.693208
.	.
.	.
.	.
Exacto	0.693147. . .

La convergencia de los valores calculados por medio de la regla trapezoidal no es particularmente rápida, pero el método es bastante simple.

Ejercicios 14.6

1. Evalúe las siguientes integrales usando la regla trapezoidal.

a. Evalúe T_0 para un panel usando la ecuación 14.6.
b. Calcule T_1 usando el valor de T_0 y la ecuación 14.11.
c. Continúe el cálculo hasta T_4.

Reúna sus resultados en forma de tabla. (Tenga cuidado: los errores en un paso se transfieren al siguiente.)

		Resultado exacto
a.	$\int_0^8 x^2 dx$	170⅔
b.	$\int_0^8 x^4 dx$	6553.6
c.	$\int_0^1 xe^{-x} dx$	1 – 2/e = 0.2642411175. . .
d.	$\int_0^{\pi/2} x \ \text{sen}(x)dx$	1.0
e.	$\int_0^1 \left(1+x^2\right)^{3/2} dx$	1.567951963. . .
f.	$\int_0^1 e^{-x^2} dx$	0.74682404. . .

14.7 LA REGLA DE SIMPSON

La regla trapezoidal se basa en aproximar la función con segmentos de línea recta. Para mejorar la precisión y la velocidad de convergencia del método, una dirección obvia a seguir sería aproximar la función con segmentos parabólicos en lugar de líneas rectas. Esta idea resulta en una aproximación a la integral conocida como **regla de Simpson**, cuyo ejemplo más simple se ilustra en la figura 14.11. Especificar en forma única una parábola requiere tres puntos, y por tanto la regla de Simpson de orden menor tiene dos paneles.

Para proceder, es necesario conocer el área bajo una parábola trazada a través de tres puntos. Hay que observar que el paso correspondiente en la derivación de la regla trapezoidal era trivial: el área bajo una línea a través de dos puntos tan sólo es $\Delta x[f(a) + f(b)]/2$.

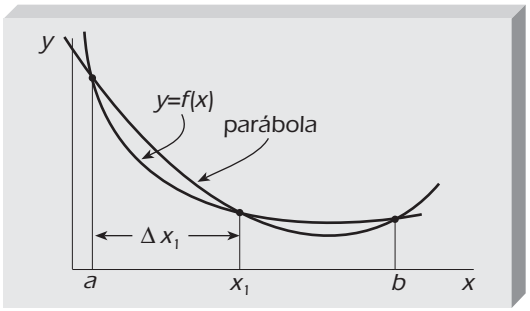


Figura 14.11 *Área bajo una parábola trazada a través de tres puntos.*

Si la curva $f(x)$ trazada en la figura 14.11 es aproximada por una parábola trazada a través de tres puntos $f(a)$, $f(b)$, y el valor de $f(x)$ en el punto intermedio del intervalo f_{mid}, puede mostrarse mediante cálculo que el área sombreada bajo la parábola, denotada como S_1, es

$$S_1 = \tfrac{1}{3}\Delta x_1\left[f(a) + 4f(a + \Delta x_1) + f(b)\right]$$

donde

$$\Delta x_1 = \frac{b-a}{2}$$

Ésta es la aproximación de la regla de Simpson de primer orden donde $k = 1$ y $n = 2^1$ paneles. El siguiente nivel de aproximación es reducir a la mitad el ancho del intervalo y dividirlo en cuatro paneles, como se muestra en la figura 14.12. El área bajo la función $f(x)$ es aproximada entonces como el área bajo las dos parábolas mostradas en la figura. Una vez más, mediante cálculo, puede mostrarse que el área bajo las dos parábolas es

$$\begin{aligned} S_2 &= \tfrac{1}{3}\Delta x_2\left\{\left[f(a) + 4f(x_1) + f(x_2)\right] + \left[f(x_2) + 4f(x_3) + f(b)\right]\right\} \\ &= \tfrac{1}{3}\Delta x_2\left\{f(a) + 4\left[f(x_1) + f(x_3)\right] + 2f(x_2) + f(b)\right\} \end{aligned} \qquad (14.12)$$

donde

$$\Delta x_2 = \frac{b-a}{2^2}$$

y

$$f_i = f\left(x = a + i\Delta x_2\right)$$

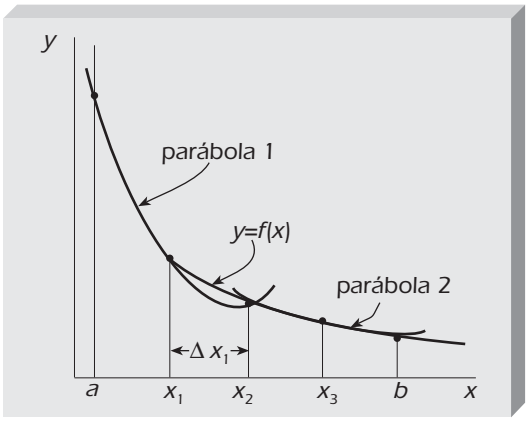


Figura 14.12 *La aproximación de la regla de Simpson de segundo orden es el área bajo dos parábolas.*

Este procedimiento puede extenderse a 8, 16, 32 o más paneles, en esa progresión. El resultado es una generalización bastante simple de la ecuación 14.12 y para $n = 2^k$ paneles es

$$S_{k=\frac{1}{3}\Delta x_k}\left[f(a) + 4\underbrace{\sum_{i=1}^{n-1} f(a+i\Delta x_k)}_{\text{sólo nones}} + 2\underbrace{\sum_{i=2}^{n-2} f(a+i\Delta x_k)}_{\text{sólo pares}} + f(b)\right] \quad (14.13)$$

La ecuación 14.13, la cual se conoce como **regla de Simpson**, es un método extremadamente popular para evaluar integrales de funciones que son uniformes, y en forma correcta. Como se verá más adelante en el capítulo, la regla de Simpson converge bien en la mayor parte de los casos y es relativamente fácil de usar. Además, la ecuación 14.13 puede adaptarse con facilidad para manejar un número non de puntos espaciados de manera desigual y es el método más común para estimar la integral de datos obtenidos en forma experimental.

Ejemplo de la regla de Simpson como una aproximación a una integral

De nuevo considere la integral

$$I = \int_1^2 (1/x)\,dx$$

Al usar primero la ecuación 14.13 para $k = 1$ se obtiene

$k = 1$

$n = 2^1 = 2$

$$\Delta x_1 = \frac{b-a}{2^2} = \frac{1}{2}$$

$S_1 = \frac{1}{3}(\frac{1}{2})[1 + 4(1/1.5) + \frac{1}{2}] = 0.6944444$

Al repetir para $k = 2$

$k = 2$
$n = 2^2 = 4$
$\Delta x_2 = ¼$
$S_2 = ⅓(¼)[1 + 4(1/1.25 + 1/1.75) + 2(1/1.5) + ½] = 0.69325397$

Continuando el cálculo, se obtienen los valores mostrados en la tabla 14.2. Para efectos de comparación, también se incluyen los resultados para la misma integral obtenida en la sección anterior usando la regla trapezoidal. Es evidente que la regla de Simpson converge mucho más rápido que la regla trapezoidal, al menos para este ejemplo.

Tabla 14.2 Resultados de la regla trapezoidal y de Simpson para la integral $\int_1^2 (1/x)\,dx$

Orden k	Número de paneles n	T_k	S_k
0	1	0.75	–
1	2	0.7083	0.6944
2	4	0.69702	0.69325
3	8	0.69412	0.69315
4	16	0.69339	0.6931466
5	32	0.693208	0.6931473
6	64	0.693162	0.6931472

Ejercicios 14.7

1. Usando la ecuación 14.13, calcule los resultados de la regla de Simpson de dos paneles y de cuatro paneles, S_1 y S_2, para las siguientes integrales.

		Resultado exacto
a.	$\int_0^8 x^2\,dx$	170⅔
b.	$\int_0^8 x^4\,dx$	6553.6
c.	$\int_0^1 xe^{-x}\,dx$	$1 - 2/e = 0.2642411175\ldots$
d.	$\int_0^{\pi/2} x \operatorname{sen}(x)\,dx$	1.0

e. $\int_0^1 \left(1+x^2\right)^{3/2} dx$ 1.567951962. . .

f. $\int_0^1 e^{-x^2} dx$ 0.74682404. . .

2. Usando la ecuación 14.13, calcule de S_1 a S_4 para las integrales mostradas en el ejercicio 1.

14.8 ERRORES COMUNES DE PROGRAMACIÓN

Al usar el método de bisección modificada para encontrar raíces pueden ocurrir dos problemas.

El primer problema es un error de redondeo, el cual puede ocurrir siempre que los valores de $f(x_1)$ y $f(x_3)$ utilizados en el cálculo sean casi iguales. El segundo problema ocurre porque no se dispone de una predicción del número exacto de iteraciones requeridas para lograr una precisión deseada. Para contrarrestar con éxito estos dos problemas, el código empleado en estos dos métodos debe detectar su ocurrencia para prevenir iteraciones excesivas y quizás infinitas.

Con respecto a la integración numérica, los tiempos de cálculo excesivos también pueden ser un problema. Esto ocurre por lo general siempre que el número de iteraciones excede de cincuenta. Con frecuencia, es posible reducir de manera significativa el tiempo de ejecución al inspeccionar con cuidado cada ciclo del programa para asegurar que sólo aquellos cálculos que deben realizarse de manera iterativa se incluyen dentro del ciclo, y mover todos los demás cálculos que se van a realizar ya sea antes o después de completarse el ciclo.

14.9 RESUMEN DEL CAPÍTULO

Todos los métodos para solucionar raíces descritos en este capítulo son de naturaleza iterativa y pueden clasificarse dentro de dos clases de algoritmos para encontrar raíces, dependiendo de si se comienza con un intervalo que contiene una raíz o con una estimación inicial de la raíz. Los procedimientos basados en bisección comienzan con un intervalo que se sabe que contiene una raíz y está garantizado que convergen dentro de un agrupamiento de tolerancia prescrito para la raíz. De los métodos de bisección, el *regula falsi* modificado es el que converge con más rapidez y se recomienda.

En algoritmos basados en el método de bisección, el intervalo inicial se depura evaluando de manera repetida la función en puntos dentro del intervalo y luego supervisando el signo de la función y determinando en cuál subintervalo se encuentra la raíz. Si los extremos izquierdo y derecho del intervalo actual son x_1 y x_3, respectivamente, el método de bisección estándar usa la función evaluada en el punto intermedio, $x_2 = \frac{1}{2}(x_1 + x_3)$. El signo de la función en x_2 se compara con el de cualquier extremo del intervalo para determinar cuál mitad del intervalo contiene la raíz. Entonces se reemplaza el intervalo entero por esta mitad y se repite el proceso. Después de n iteraciones, la raíz estará contenida en un intervalo de tamaño $(x_3 - x_1)/2n$.

En el método *regula falsi*, las condiciones son las mismas que para el método de bisección. Pero en lugar de usar el punto intermedio del intervalo, se usa una línea recta que conecta los puntos en los extremos del intervalo para interpolar la posición de la raíz. La intersección de esta línea con el eje x determina el valor de x_2 que se usará en el siguiente paso. Este valor de x_2 está dado por la ecuación

$$x_2 = x_1 - (x_3 - x_1)\frac{f(x_1)}{f(x_3) - f(x_1)}$$

en lugar de la ecuación para el punto intermedio. La convergencia es más rápida que con el método de bisección. Sin embargo, es probable que el intervalo converja con la raíz sólo en un lado.

El método *regula falsi* modificado es igual que el método *regula falsi* con excepción del siguiente cambio: en cada iteración, cuando el intervalo completo es reemplazado con el subintervalo que contiene la raíz, se usa un factor de relajación para modificar primero el valor de la función en el extremo fijo del subintervalo. Se sugiere un factor de relajación de aproximadamente 0.9. Esta característica adicional causará que el intervalo converja desde ambos extremos, y entonces la convergencia se basa en el tamaño del intervalo. Este método es el procedimiento preferido para encontrar una raíz de una función que no es muy difícil evaluar y que se sabe que tiene una raíz en un intervalo especificado.

El método de la secante reemplaza a la función por una línea secante a través de dos puntos y luego encuentra el punto de intersección de la línea con el eje x. El algoritmo requiere dos números de entrada, x_0 y Δx_0, correspondientes a suposiciones iniciales para la raíz y para un intervalo que contenga la raíz. Este par de valores es reemplazado luego con el par $(x_1, \Delta x_1)$ donde

$$x_1 = x_0 + \Delta x_0$$

y

$$\Delta x_1 = \frac{f(x_1)}{f(x_0) - f(x_1)}\,\Delta x_0$$

y el proceso se continúa hasta que el nuevo intervalo Δx es suficientemente pequeño.

Los métodos para resolver raíces son susceptibles de codificación en C++. Sin embargo, el éxito de un programa para encontrar la raíz de una función por lo general depende de la calidad de la información suministrada por el usuario; es decir, qué tan precisa es la suposición inicial o el intervalo de búsqueda y qué tan bien corresponde el método elegido a las circunstancias del problema. Los problemas en tiempo de ejecución se rastrean con más frecuencia hasta errores en la codificación de la función o a un diagnóstico inadecuado suministrado por el usuario para problemas potenciales.

La integral de $f(x)$ desde $x = a$ hasta $x = b$, escrita como

$$I = \int_a^b f(x)\,dx$$

se evalúa de manera numérica al calcular el área bajo la curva $f(x)$ a través del rango de x especificado. Los procedimientos para estimar esta área consisten en dividir el intervalo $a \le x \le b$ en n paneles de ancho $\Delta x_i (i = 1,n)$, y aproximando la función $f(x)$ a través de cada panel por una función más simple.

La regla trapezoidal resulta de reemplazar la función $f(x)$ con segmentos de línea recta a través de los paneles Δx_i. El valor aproximado para la integral está dado entonces por la siguiente fórmula, la cual se conoce como la regla trapezoidal:

$$\int_a^b f(x)\,dx \approx \tfrac{1}{2}\Delta x_n\left(f(a) + 2\sum_{i-1}^{n-1} f_i + f(b)\right)$$

Si los paneles son de tamaño igual y el número de paneles es $n = 2^k$, donde k es un número entero positivo, entonces la aproximación por la regla trapezoidal se denomina T_k y satisface la ecuación

$$T_k = \tfrac{1}{2}T_{k-1} + \Delta x_k \sum_{\substack{i=1 \\ \text{sólo nones}}}^{n-1} f(a + i\Delta x_k)$$

donde

$$\Delta x_k = \frac{b-a}{2^k}$$

En el siguiente nivel de aproximación la función $f(x)$ se reemplaza por $n/2$ segmentos parabólicos en los pares de paneles de tamaño igual, $\Delta x = (b - a)/n$ y resulta en la fórmula para el área conocida como la regla de Simpson:

$$\int_a^b f(x)\,dx \approx \tfrac{1}{3}\Delta x_k\left[f(a) + 4\sum_{\substack{i=1 \\ \text{sólo nones}}}^{n-1} f(a + i\Delta x_k) + 2\sum_{\substack{i=2 \\ \text{sólo pares}}}^{n-2} f(a + i\Delta x_k) + f(b)\right]$$

CAPÍTULO 15
Operaciones bit por bit

TEMAS

15.1 EL OPERADOR AND

15.2 EL OPERADOR OR INCLUSIVO

15.3 EL OPERADOR OR EXCLUSIVO

15.4 EL OPERADOR DE COMPLEMENTO

15.5 ELEMENTOS DE DATOS DE TAMAÑO DIFERENTE

15.6 LOS OPERADORES DE DESPLAZAMIENTO

15.7 RESUMEN DEL CAPÍTULO

C++ opera con entidades de datos que se almacenan como uno o más bytes, como constantes y variables de carácter, número entero y número de precisión doble. Además, C++ prevé la manipulación de bits individuales de valores y variables de carácter y números enteros. Los operadores que se usan para ejecutar manipulaciones de bits se llaman operadores de bits y se enlistan en la tabla 15.1.

Todos los operadores mostrados en la tabla 15.1, excepto ~, son operadores binarios, que requieren dos operandos. Cada operando es tratado como un número binario consistente en una serie de unos y ceros individuales. Los bits respectivos en cada operando se comparan bit por bit y el resultado se determina con base en la operación seleccionada.

Tabla 15.1 Operadores de bits

Operador	Descripción
&	AND bit por bit
|	OR inclusivo bit por bit
^	OR exclusivo bit por bit
~	Complemento de uno bit por bit
<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha

15.1 EL OPERADOR AND

El operador AND causa una comparación AND bit por bit entre sus dos operandos. *El resultado de cada comparación bit por bit es un 1 sólo cuando ambos bits comparados son 1; de lo contrario, el resultado de la operación AND es un 0*. Por ejemplo, suponga que los siguientes dos números de ocho bits se van a procesar usando AND:

```
1 0 1 1 0 0 1 1
1 1 0 1 0 1 0 1
- - - - - - - -
```

Para realizar una operación AND, cada bit en un operando es comparado con el bit que ocupa la misma posición en el otro operando. La figura 15.1 ilustra la correspondencia entre bits para estos dos operandos. Como se muestra en la figura, cuando ambos bits que se comparan son 1, el resultado es 1, de lo contrario el resultado es 0. El resultado de cada comparación, por supuesto, es independiente de cualquier otra comparación de bits.

```
  1 0 1 1 0 0 1 1
& 1 1 0 1 0 1 0 1
  - - - - - - - -
  1 0 0 1 0 0 0 1
```

Figura 15.1 *Una muestra de la operación AND.*

El programa 15.1 ilustra el uso de una operación AND. En este programa, la variable `op1` se inicializa con el valor octal 325, el cual es el equivalente octal del número binario 1 1 0 1 0 1 0 1, y la variable `op2` se inicializa con el valor octal 263, el cual es la representación octal del número binario 1 0 1 1 0 0 1 1. Éstos son los mismos dos números binarios ilustrados en la figura 15.1.

Programa 15.1

```
#include <iostream>
using namespace std;

int main()
{
  int op1 = 0325, op2 = 0263;

  int op3 = op1 & op2;
  cout << oct << op1 << "procesado en AND con " << op2 << " es " << op3 << endl;

  return 0;
}
```

El programa 15.1 produce la siguiente salida:

```
325 procesado en AND con 263 es 221
```

El resultado de procesar los números octales 325 y 263 con AND es el número octal 221. El equivalente binario de 221 es el número binario 1 0 0 1 0 0 0 1, el cual es el resultado de la operación AND ilustrada en la figura 15.1.

Las operaciones AND son útiles en extremo para enmascarar, o eliminar, bits seleccionados de un operando. Éste es un resultado directo del hecho que procesar con AND cualquier bit (1 o 0) con un 0 obliga al bit resultante a ser un 0, mientras que procesar con AND cualquier bit (1 o 0) con un 1 deja sin cambios al bit original. Por ejemplo, suponga que la variable `op1` tiene el patrón de bits arbitrario x x x x x x x x, donde cada x puede ser 1 o 0, independiente de cualquier otra x en el número. El resultado de procesar con AND este número binario con el número binario 0 0 0 0 1 1 1 1 es:

```
        op1 =   x x x x x x x x
        op2 =   0 0 0 0 1 1 1 1
                - - - - - - - -
  Resultado =   0 0 0 0 x x x x
```

Como se puede observar en este ejemplo, los ceros en `op2` efectivamente enmascaran, o eliminan, los bits respectivos en `op1`, mientras los unos en `op2` filtran, o pasan, los bits respectivos en `op1` sin alterar sus valores. En este ejemplo, la variable `op2` se llama **máscara**. Al elegir la máscara de manera apropiada, puede seleccionarse, o filtrarse, cualquier bit individual en un operando para su inspección. Por ejemplo, procesar con AND la variable `op1` con la máscara 0 0 0 0 0 1 0 0 obliga a todos los bits del resultado a ser 0, excepto el tercer bit. El tercer bit del resultado será una copia del tercer bit de `op1`. Por tanto, si el resultado del AND es 0, el tercer bit de `op1` debe haber sido un 0, y si el resultado del AND es un número diferente de cero, el tercer bit debe haber sido un 1.

El programa 15.2 utiliza esta propiedad de enmascaramiento para convertir las letras minúsculas en su forma mayúscula, suponiendo que las letras son almacenadas usando el código ASCII. El algoritmo para convertir letras se basa en el hecho que los códigos binarios para las letras minúsculas y mayúsculas en ASCII son los mismos excepto por el bit

cinco, el cual es 1 para las letras minúsculas y 0 para las letras mayúsculas.[1] Por ejemplo, el código binario para la letra a es 01100001 (hex 61), mientras el código binario para la letra A es 01000001 (hex 41). Del mismo modo, el código binario para la letra z es 01111010 (hex 7A), mientras el código binario para la letra Z es 01011010 (hex 5A). (Véase el apéndice B para los valores hexadecimales de las letras mayúsculas y minúsculas.) Por tanto, dada una letra minúscula, puede convertirse en su forma mayúscula al obligar al quinto bit a ser 0. Esto se logra en el programa 15.2 al enmascarar el código de la letra con el valor binario 11011111, el cual tiene el valor hexadecimal DF.

Programa 15.2

```
#include <iostream>
using namespace std;

const int A_MAYUS = 0xDF;
void mayus(char *);  // prototipo de la función

int main()
{
  char palabra[81];   // suficiente almacenamiento para una línea completa

  cout << "Introduzca una cadena de letras mayusculas y minusculas:\n";
  cin.getline(palabra,80,'\n');
  cout << "\nLa cadena de letras que se acaba de introducir es:\n"
       << palabra << endl;
  mayus(palabra);
  cout << "\nEsta cadena, en letras mayusculas es:\n"
       << palabra << endl;
}
void mayus(char *palabra)
{
  while (*palabra != '\0')
    *palabra++ &= A_MAYUS;
}
```

A continuación se presenta una muestra de la ejecución del programa 15.2:

```
Introduzca una cadena de letras mayusculas y minusculas:
abcdefgHIJKLMNOPqrstuvwxyz

La cadena de letras que se acaba de introducir es:
abcdefgHIJKLMNOPqrstuvwxyz

Esta cadena, en letras mayusculas es:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

[1]Esto supone el esquema de numeración convencional que empieza con el bit 0 como el bit que se encuentra a la extrema derecha. Usando esta convención el bit de la extrema derecha (o bit 0) se conoce como el bit menos significativo (LSB, por sus siglas en inglés) y el bit de la extrema izquierda se conoce como el bit más significativo (MSB, por sus siglas en inglés). Aquí el MSB es el bit 7.

Al revisar el programa 15.2, se puede observar primero que la cadena introducida se ha almacenado y transmitido a `mayus()` como una cadena C, la cual es un arreglo de caracteres. Hacer esto permite a la función recibir y operar en los valores de carácter originales en lugar de recibir una copia de estos valores. Segundo, hay que observar que las letras minúsculas son convertidas a su forma mayúscula, mientras las letras mayúsculas permanecen inalteradas. Esto se debe a que el bit cinco de todas las letras mayúsculas es 0 para empezar, así que forzar este bit a ser 0 usando la máscara no tiene efecto. Sólo cuando el bit cinco es un 1, como lo es en las letras minúsculas, se altera el carácter introducido.

15.2 EL OPERADOR OR INCLUSIVO

El operador OR inclusivo, |, ejecuta una comparación bit por bit de sus dos operandos en una forma similar a la del AND bit por bit. El resultado de la comparación OR, sin embargo, es determinada por la siguiente regla:

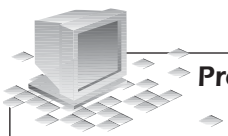
El resultado de la comparación es 1 si cualquier bit comparado es un 1, de lo contrario el resultado es un 0.

La figura 15.2 ilustra una operación OR. Como se muestra en la figura, cuando cualquiera de los dos bits que se están comparando es un 1, el resultado es un 1; de lo contrario el resultado es un 0. Como con todas las operaciones con bits, el resultado de cada comparación es, por supuesto, independiente de cualquier otra comparación.

```
  1 0 1 1 0 0 1 1
| 1 1 0 1 0 1 0 1
  - - - - - - - -
  1 1 1 1 0 1 1 1
```

Figura 15.2 *Una muestra de operación OR.*

El programa 15.3 ilustra una operación OR, usando los valores octales de los operandos ilustrados en la figura 15.2.



Programa 15.3

```
#include <iostream>
using namespace std;

int main()
{
  int op1 = 0325, op2 = 0263;

  int op3 = op1 | op2;
  cout << oct << op1 << " procesado en OR con " << op2 << " es " << op3 << endl;
}
```

El programa 15.3 produce la siguiente salida:

```
325 procesado en OR con 263 es 367
```

El resultado de procesar en OR los números octales 325 y 263 es el número octal 367. El equivalente binario de 367 es 1 1 1 1 0 1 1 1, el cual es el resultado de la operación OR ilustrada en la figura 15.2.

Las operaciones OR inclusivas son muy útiles para forzar a los bits seleccionados a adoptar el valor de 1 o para dejar sin cambio otros valores de bits. Esto es un resultado directo del hecho que procesar con OR cualquier bit (1 o 0) con un 1 obliga al bit resultante a ser un 1, mientras que procesar con OR cualquier bit (1 o 0) con un 0 deja al bit original sin cambios. Por ejemplo, suponga que la variable `op1` tiene el patrón de bits arbitrario x x x x x x x x, donde cada x puede ser 1 o 0, independiente de cualquier otra *x* en el número. El resultado de procesar con OR este número binario con el número binario 1 1 1 1 0 0 0 0 es

```
        op1 =   x x x x x x x x
        op2 =   1 1 1 1 0 0 0 0
                - - - - - - - -
  Resultado =   1 1 1 1 x x x x
```

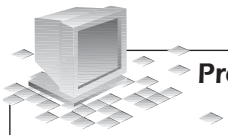
Como se puede observar en este ejemplo, los unos en `op2` obligan a los bits resultantes a ser 1, mientras los ceros en `op2` filtran, o dejan pasar, los bits respectivos en `op1` sin cambio en sus valores. Por tanto, usando una operación OR puede producirse una operación de enmascaramiento similar, como con la operación AND, con la excepción que los bits enmascarados se establecen en 1 en lugar de en 0. Otra forma de expresar esto es decir que procesar en OR con un 0 tiene el mismo efecto que procesar en AND con un 1.

El programa 15.4 usa esta propiedad de enmascaramiento para convertir letras mayúsculas en una palabra en su respectiva forma minúscula, suponiendo que las letras están almacenadas usando el código ASCII. El algoritmo para convertir letras es similar al usado en el programa 15.2, y convierte letras mayúsculas en su forma minúscula forzando al quinto bit en cada letra a ser 1. Esto se logra en el programa 15.4 enmascarando el código de la letra con el valor binario 00100000, el cual tiene el valor hexadecimal 20.

A continuación se presenta una muestra de la ejecución del programa 15.4:

```
Introduzca una cadena de letras mayusculas y minusculas:
abcdefgHIJKLMNOPqrstuvwxyz

La cadena de letras que se acaba de introducir es:
abcdefghijklmnopqrstuvwxyz
Esta cadena, en letras minúsculas es:
abcdefghijklmnopqrstuvwxyz
```

Al revisar el programa 15.4, se puede observar primero que la cadena introducida se ha almacenado y transmitido a `minusc()` como una cadena C, la cual es un arreglo de caracteres. Esto permite que la función llamada reciba y procese los valores de caracteres originales en lugar de una copia de los valores. En segundo lugar, se puede observar que las letras mayúsculas son convertidas a una forma minúscula, mientras las letras mayúsculas permanecen inalteradas. Esto se debe a que el bit cinco de todas las letras minúsculas es 1 para comenzar, así que forzar este bit a ser uno usando la máscara no tiene efecto. Sólo cuando el bit cinco es un 0, como sucede en las letras mayúsculas, se altera el carácter introducido.



Programa 15.4

```
#include <iostream>
using namespace std;

const int A_MINUS = 0x20;
void minus(char *);   // prototipo de la función

int main()
{
  char palabra[81];    // suficiente almacenamiento para una línea completa

  cout << "Introduzca una cadena de letras mayusculas y minusculas:\n";
  cin.getline(palabra,80,'\n');
  cout << "\nLa cadena de letras que se acaba de introducir es:\n"
       << palabra << endl;
  minus(palabra);
  cout << "\nEsta cadena, en letras minusculas es:\n"
       << palabra << endl;
}
void minus(char *palabra)
{
  while (*palabra != '\0')
    *palabra++ |= A_MINUS;
}
```

15.3 El operador OR exclusivo

El operador OR exclusivo, ^, realiza una comparación bit por bit de sus dos operandos. El resultado de la comparación es determinada por la siguiente regla:

El resultado de la comparación es 1 si uno y sólo uno de los bits que se están comparando es un 1, de lo contrario el resultado es 0.

La figura 15.3 ilustra una operación OR exclusiva. Como se muestra en la figura, cuando ambos bits que se están comparando tienen el mismo valor (ambos son 1 o ambos son 0), el resultado es un 0. Sólo cuando ambos bits tienen valores diferentes (un bit es un 1 y el otro un 0) el resultado es un 1. Una vez más, cada par o comparación de bits es independiente de cualquier otra comparación de bits.

```
  1 0 1 1 0 0 1 1
^ 1 1 0 1 0 1 0 1
  - - - - - - - -
  0 1 1 0 0 1 1 0
```

Figura 15.3 Una muestra de la operación OR exclusiva.

Una operación OR exclusiva puede usarse para crear el valor opuesto, o complemento, de cualquier bit individual en una variable. Esto es un resultado directo del hecho que procesar en OR exclusivo cualquier bit (1 o 0) con un 1 obliga al bit resultante a ser del valor opuesto a su estado original, mientras procesar en OR exclusivo cualquier bit (1 o 0) con un 0 deja al bit original inalterado. Por ejemplo, suponga que la variable `op1` tiene el patrón de bits arbitrario x x x x x x x x, donde cada x puede ser 1 o 0, independiente de cualquier otra x en el número. Usando la notación en que $\overline{x}$ es el valor complementario (opuesto) de x, el resultado de procesar en OR exclusivo este número binario con el número binario 0 1 0 1 0 1 0 1 es:

```
        op1 =   x x x x x x x x
        op2 =   0 1 0 1 0 1 0 1
                - - - - - - - -

                _   _   _   _
  Resultado =   x x x x x x x x
```

Como se puede observar en este ejemplo, los unos en `op2` obligan a los bits resultantes a ser el complemento de sus valores de bits originales, mientras los ceros en `op2` filtran, o dejan pasar, los bits respectivos en `op1` sin cambio en sus valores.

Muchos métodos de encriptamiento usan la operación OR exclusiva para codificar los datos procesando en OR cada carácter en la cadena con un valor máscara. La elección del valor de la máscara, el cual se conoce como la clave para encriptar, es arbitraria y puede usarse cualquier valor clave.

El programa 15.5 usa una clave para encriptar de 52 para codificar un mensaje introducido por el usuario.

A continuación se presenta una muestra de la ejecución del programa 15.5.

```
Introduzca un enunciado:
Buenos dias

El enunciado que se acaba de introducir es:
Buenos dias

La version encriptada de este enunciado es:
s[[P¶Y[FZ]ZS
```

Decodificar un mensaje encriptado requiere que se procese con OR exclusivo el mensaje codificado usando la clave de encriptamiento original, lo cual es un ejercicio para resolver en casa.

Programa 15.5

```
#include <iostream>
using namespace std;

void encript(char *); // prototipo de la funcion

int main()
{
  char mensaje[81];   // suficiente almacenamiento para una linea completa

  cout << "Introduzca un enunciado:\n";
  cin.getline(mensaje,80,'\n');
  cout << "\n El enunciado que se acaba de introducir es:\n"
       << mensaje << endl;
  encript(mensaje);
  cout << "\n La version encriptada de este enunciado es:\n"
       << mensaje << endl;
}

void encript(char *mensaje)
{
  while (*mensaje != '\0')
    *mensaje++ ^= 52;
}
```

15.4 EL OPERADOR DE COMPLEMENTO

El operador de complemento, ~, es un operador unitario que cambia cada bit 1 en su operando a 0 y cada bit 0 a 1. Por ejemplo, si la variable `op1` contiene el número binario 11001010, `~op1` reemplaza este número binario con el número 00110101. El operador de complemento se usa para forzar a cualquier bit en un operando a 0, independiente del número de bits real usado para almacenar el número. Por ejemplo, la instrucción

```
op1 = op1 & ~07;    // 07 es un número octal
```

o su forma más corta,

```
op1 &= ~07; // 07 es un número octal
```

establecen los últimos tres bits de `op1` en cero, sin importar cómo se almacene `op1` dentro de la computadora. Por supuesto, cualquiera de estas dos instrucciones se puede reemplazar empleando la operación AND con ceros en los tres últimos bits de `op1`, si se conoce

el número de bits usado para almacenar `op1`. En una computadora que usa 16 bits para almacenar números enteros, la operación AND apropiada es

```
op1 = op1 & 0177770;    // en octal
```

o

```
op1 = op1 & 0xFFF8;    // en hexadecimal
```

Para una computadora que usa 32 bits para almacenar números enteros, el AND anterior establece también los 16 bits de la extrema izquierda o de orden superior en cero, lo cual es un resultado no intencional. La instrucción correcta para 32 bits es:

```
op1 = op1 & 027777777770;    // en octal
```

o

```
op1 = op1 & 0xFFFFFFF8;     // en hexadecimal
```

Usar el operador de complemento en esta situación libera al programador de tener que determinar el tamaño de almacenamiento del operando y, de mayor importancia, hace que el programa sea transferible entre máquinas que usan diferentes tamaños de almacenamiento de números enteros.

15.5 ELEMENTOS DE DATOS DE TAMAÑO DIFERENTE

Cuando los operadores de bits `&`, `|`, y `^` se usan con operandos de diferentes tamaños, siempre se incrementa el tamaño de bits del operando más corto para igualar el tamaño del operando más grande. La figura 15.4 ilustra la extensión de un número entero sin signo de 16 bits en un número de 32 bits.

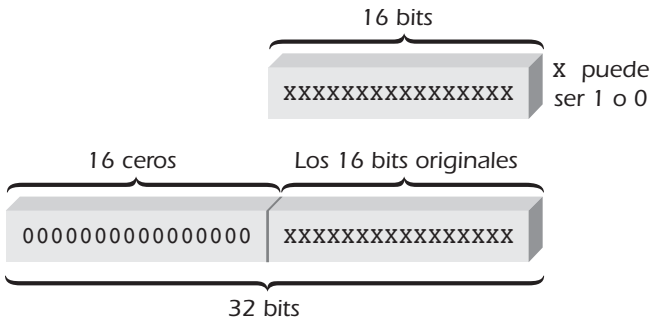


Figura 15.4 *Extensión de datos sin signo de 16 bits a 32 bits.*

Como muestra la figura, los bits adicionales se agregan a la izquierda del número original y se llenan con ceros. Esto es equivalente a añadirle al número ceros a la izquierda, lo cual no tiene efecto en el valor del número.

Cuando se extienden números con signo, el bit original a la extrema izquierda es reproducido en los bits adicionales que se agregan al número. Como se ilustra en la figura 15.5, si el bit original a la extrema izquierda es 0, correspondiente a un número positivo, se coloca 0 en cada una de las posiciones de bit adicionales. Si el bit de la extrema izquierda

es 1, lo cual corresponde a un número negativo, se coloca 1 en las posiciones de bit adicionales. En cualquier caso, el número binario resultante tiene el mismo signo y magnitud del número original.

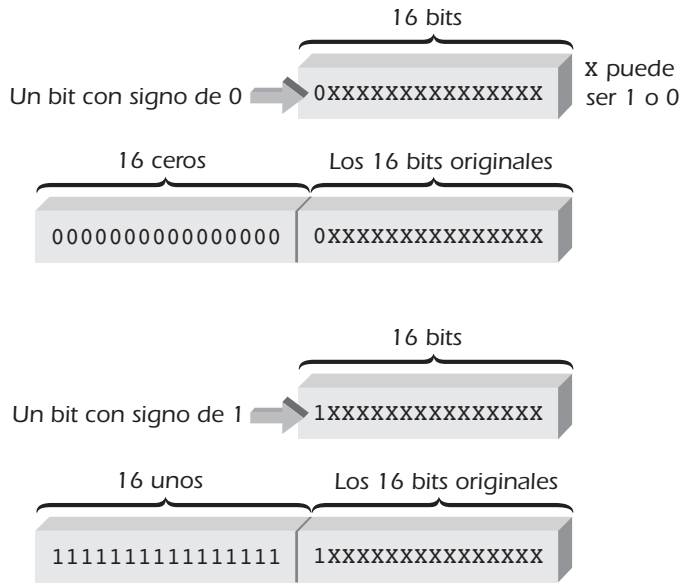


Figura 15.5 *Extensión de datos de 16 bits con signo a 32 bits.*

15.6 LOS OPERADORES DE DESPLAZAMIENTO

El operador de desplazamiento a la izquierda, <<, causa que los bits en un operando sean desplazados a la izquierda por una cantidad determinada. Por ejemplo, la instrucción

```
op1 = op1 << 4;
```

causa que los bits en `op1` sean desplazados cuatro bits a la izquierda, llenando cualesquiera bits vacantes con un cero. La figura 15.6 ilustra el efecto de cambiar el número binario 1111100010101011 a la izquierda por cuatro posiciones de bit.

Para los números enteros sin signo, cada desplazamiento a la izquierda corresponde a una multiplicación por dos. Esto sucede también con números con signo que usan la representación complementaria a dos, en tanto el bit de la extrema izquierda no cambie de valor. Dado que un cambio en el bit de la extrema izquierda de un número en complemento a dos representa un cambio tanto en el signo como en la magnitud representada por el bit, dicho cambio no representa una simple multiplicación por dos.

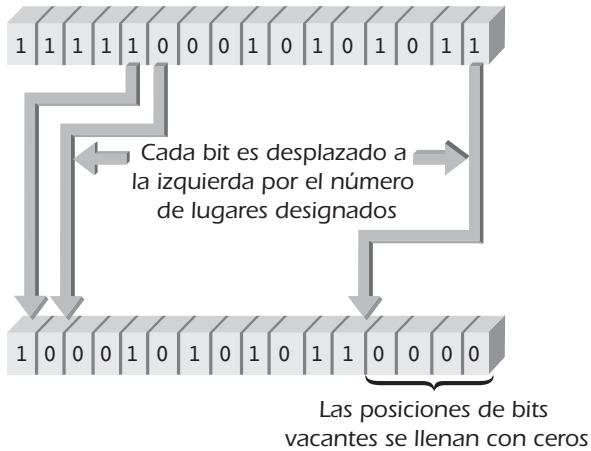


Figura 15.6 *Un ejemplo de un desplazamiento a la izquierda.*

El operador de desplazamiento a la derecha, >>, hace que los bits en un operando sean desplazados a la derecha por una cantidad determinada. Por ejemplo, la instrucción

```
op2 = op1 >> 3;
```

causa que los bits en `op1` sean desplazados a la derecha tres posiciones de bit. La figura 15.7a ilustra el desplazamiento a la derecha del número binario sin signo 1111100010101011 por tres posiciones de bit. Como se ilustra, los tres bits de la extrema derecha son desplazados más allá del final y se pierden.

Para números sin signo, el bit de la extrema izquierda no se utiliza como un bit con signo. Para este tipo de número, los bits de la extrema izquierda vacantes siempre se llenan con ceros. Éste es el caso que se ilustra en la figura 15.7a.

Para números con signo, lo que se llena en los bits vacantes depende de la computadora. La mayor parte de las computadoras reproduce el bit con signo original del número. La figura 15.7b ilustra el desplazamiento a la derecha de un número binario negativo por cuatro posiciones de bit, donde el bit con signo es reproducido en los bits vacantes. La figura 15.7c ilustra el desplazamiento a la derecha equivalente de un número binario con signo positivo.

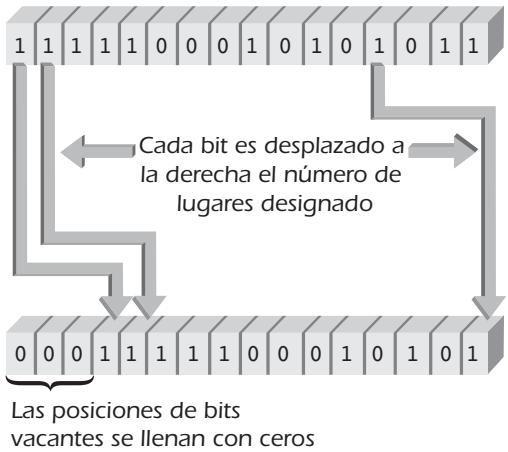


Figura 15.7a *Un desplazamiento aritmético a la derecha sin signo.*

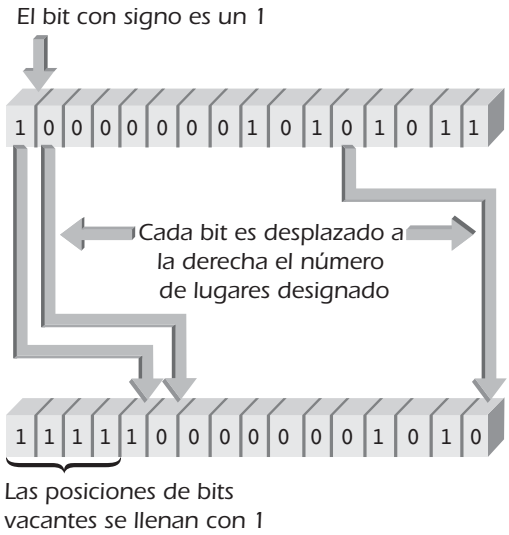


Figura 15.7b *El desplazamiento a la derecha de un número binario negativo.*

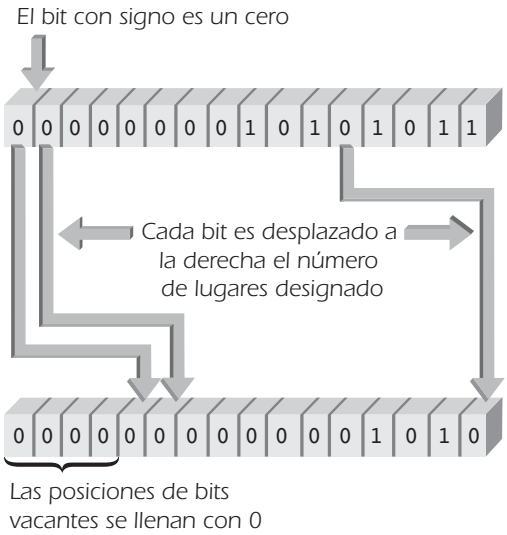


Figura 15.7c *El desplazamiento a la derecha de un número binario positivo.*

El tipo de relleno ilustrado en las figuras 15.7b y c, donde el bit con signo es reproducido en posiciones de bits vacantes, se llama ***desplazamiento aritmético a la derecha***. En un desplazamiento aritmético a la derecha, cada desplazamiento individual a la derecha corresponde a una división entre dos.

En vez de reproducir el bit con signo en números con signo con desplazamiento a la derecha, algunas computadoras llenan de manera automática los bits vacantes con ceros. Este tipo de desplazamiento se llama ***desplazamiento lógico***. Para números con signo positivos, donde el bit de la extrema izquierda es 0, tanto el desplazamiento aritmético a la derecha como el desplazamiento lógico producen el mismo resultado. Los resultados de estos dos desplazamientos sólo son diferentes cuando están implicados números negativos.

Ejercicios para el capítulo 15

1. Determine los resultados de las siguientes operaciones:

```
a.   11001010   b.   11001010   c.   11001010
   & 10100101      | 10100101      ^ 10100101
     --------        --------        --------
```

2. Escriba las representaciones octales de los números binarios dados en el ejercicio 1.
3. Determine los resultados octales de las siguientes operaciones, suponiendo números sin signo:
   a. el número octal 0157 desplazado a la izquierda una posición de bit
   b. el número octal 0701 desplazado a la izquierda dos posiciones de bit
   c. el número octal 0673 desplazado a la derecha dos posiciones de bit
   d. el número octal 067 desplazado a la derecha tres posiciones de bit
4. Repita el ejercicio 3 suponiendo que los números son tratados como valores con signo.
5. a. Suponga que el patrón de bits arbitrario xxxxxxxx, donde cada x puede representar 1 o 0, está almacenado en la variable en número entero llamada `bandera`.

Determine el valor octal de una máscara que pueda ser procesada con AND con el patrón de bits para reproducir el tercer y cuarto bits de `bandera` y establecer todos los otros bits en cero. El bit de la extrema derecha en `bandera` es considerado el bit 0.

b. Determine el valor octal de una máscara que pueda ser procesada con OR inclusivo con el patrón de bits en `bandera` para reproducir el tercer y cuarto bits de `bandera` y establecer todos los otros bits en 1. Una vez más, considere al bit de la extrema derecha en `bandera` como el bit 0.

c. Determine el valor octal de una máscara que pueda utilizarse para complementar los valores del tercer y cuarto bits de `bandera` y dejar todos los otros bit inalterados. Determine la operación de bit que debería usarse con el valor de máscara para producir el resultado deseado.

6. a. Escriba la forma de complemento a dos del número decimal −1, usando ocho bits. (*Sugerencia*: Remítase a la sección 1.6 para una revisión de los números de complemento a dos.)

b. Repita el ejercicio 6a usando 16 bits para representar el número decimal −1 y compare su respuesta con su respuesta anterior. ¿La versión de 16 bits podría haberse obtenido extendiendo el signo de la versión de 8 bits?

7. Como se señaló en el texto, el programa 15.2 no tiene efecto en las letras mayúsculas. Usando los códigos ASCII mostrados en el apéndice B, determine cuáles otros caracteres no serían afectados por el programa 15.2.

8. Modifique el programa 15.2 de modo que pueda leerse un enunciado completo y convertirse en valores de letras minúsculas. (*Sugerencia*: Cuando un espacio es enmascarado por el programa 15.2, el carácter resultante es \0, el cual termina la salida.)

9. Modifique el programa 15.4 para permitir que se introduzca un enunciado completo y se convierta en letras mayúsculas. Asegúrese que su programa no altere ningún otro carácter o símbolo introducidos.

10. Modifique el programa 15.5 para permitir que la clave de encriptación sea un valor introducido por el usuario.

11. Modifique el programa 15.5 para hacer que su salida se escriba en un archivo llamado `codificado.dat`.

12. Escriba un programa en C++ que lea el enunciado encriptado producido por el programa escrito para el ejercicio 10, decodifique el enunciado e imprima los valores decodificados en el dispositivo de salida estándar de su sistema.

13. Escriba un programa en C++ que despliegue los primeros ocho bits de cada valor de carácter introducido en una variable llamada `ch`. (*Sugerencia:* Suponiendo que cada carácter se almacena usando ocho bits, empiece por usar la máscara hexadecimal 80, la cual corresponde al número binario 10000000. Si el resultado de la operación de enmascaramiento es un 0, despliegue un 0; de lo contrario despliegue un 1. Luego cambie la máscara un lugar a la derecha para examinar el siguiente bit, y así hasta que se hayan procesado todos los bits en la variable `ch`.)

14. Escriba un programa en C++ que invierta los bits en una variable en número entero llamada `bien` y almacene los bits invertidos en la variable llamada `bien_inv`. Por ejemplo, si el patrón de bits 11100101, correspondiente al número octal 0345,

se asigna a `bien`, el patrón de bits 10100111, correspondiente al número octal 0247, debería producirse y almacenarse en `bien_inv`.

15.7 RESUMEN DEL CAPÍTULO

1. Los bits individuales de variables y constantes de carácter y de números enteros pueden manipularse usando operadores de bits de C++. Éstos son los operadores AND, OR inclusivo, OR exclusivo, complemento a uno, desplazamiento a la izquierda y desplazamiento a la derecha.
2. Los operadores AND y OR inclusivo son útiles para crear máscaras. Estas máscaras pueden usarse para transmitir o eliminar bits individuales del operando seleccionado. El operador OR exclusivo es útil para complementar los bits de un operando.
3. Cuando se usan los operadores AND y OR con operandos de diferentes tamaños, el operando más corto siempre es incrementado en su tamaño de bits para igualar el tamaño del operando más grande.
4. Los operadores de desplazamiento producen diferentes resultados dependiendo de si el operando es un valor con signo o sin signo.

APÉNDICE A
Tabla de precedencia de operadores

La tabla A.1 presenta los símbolos, precedencia, descripciones y asociatividad de los operadores de C++. Los operadores hacia la parte superior de la tabla tienen una precedencia mayor que aquellos que se encuentran hacia la parte inferior. Los operadores dentro de cada casilla tienen la misma precedencia y asociatividad.

Tabla A.1 Resumen de operadores de C++

Operador	Descripción	Asociatividad

`()`
`[]`
`->`
`.`
	Llamada a función
	Elemento de un arreglo
	Referencia al apuntador de miembro en una estructura
	Referencia a un miembro de una estructura
		Izquierda a derecha

`++`
`--`
`-`
`!`
`~`
`(type)`
`sizeof`
`&`
`*`
	Incremento
	Decremento
	Menos unario
	Negación lógica
	Complemento a uno
	Conversión de tipo (molde)
	Tamaño del almacenamiento
	La dirección de
	Indirección
		Derecha a izquierda

`*`
`/`
`%`
	Multiplicación
	División
	Módulo (residuo)
		Izquierda a derecha

`+` `-`	Adición Sustracción	Izquierda a derecha
`<<` `>>`	Desplazamiento a la izquierda Desplazamiento a la derecha	Izquierda a derecha

`<`
`<=`
`>`
`>=`
	Menor que
	Menor que o igual a
	Mayor que
	Mayor que o igual a
		Izquierda a derecha

Tabla A.1 Resumen de operadores C++ (continución)

Operador	Descripción	Asociatividad
`==` `!=`	Igual a Diferente a	Izquierda a derecha
`&`	AND bit por bit	Izquierda a derecha
`^`	OR exclusivo bit por bit	Izquierda a derecha
`|`	OR inclusivo bit por bit	Izquierda a derecha
`&&`	AND lógico	Izquierda a derecha
`||`	OR lógico	Izquierda a derecha
`?:`	Expresión condicional	Derecha a izquierda

`=`
`+= –= *=`
`/= %= &=`
`^= |=`
`<<= >>=`
	Asignación
	Asignación
	Asignación
	Asignación
	Asignación
		Derecha a izquierda

`,`	Coma	Izquierda a derecha

APÉNDICE B

Códigos ASCII de caracteres

Tecla(s)	Dec	Oct	Hex	Tecla(s)	Dec	Oct	Hex	Tecla(s)	Dec	Oct	Hex
Ctrl 1	0	0	0	Esc	27	33	1B	6	54	66	36
Ctrl A	1	1	1	Ctrl <	28	34	1C	7	55	67	37
Ctrl B	2	2	2	Ctrl /	29	35	1D	8	56	70	38
Ctrl C	3	3	3	Ctrl =	30	36	1E	9	57	71	39
Ctrl D	4	4	4	Ctrl –	31	37	1F	:	58	72	3A
Ctrl E	5	5	5	Space	32	40	20	;	59	73	3B
Ctrl F	6	6	6	!	33	41	21	<	60	74	3C
Ctrl G	7	7	7	"	34	42	22	=	61	75	3D
Ctrl H	8	10	8	#	35	43	23	>	62	76	3E
Ctrl I	9	11	9	$	36	44	24	?	63	77	3F
Ctrl J (If)	10	12	A	%	37	45	25	@	64	100	40
Ctrl K	11	13	B	&	38	46	26	A	65	101	41
Ctrl L	12	14	C	'	39	47	27	B	66	102	42
Ctrl M (Ret)	13	15	D	(	40	50	28	C	67	103	43
Ctrl N	14	16	E	)	41	51	29	D	68	104	44
Ctrl O	15	17	F	*	42	52	2A	E	69	105	45
Ctrl P	16	20	10	+	43	53	2B	F	70	106	46
Ctrl Q	17	21	11	,	44	54	2C	G	71	107	47
Ctrl R	18	22	12	–	45	55	2D	H	72	110	48
Ctrl S	19	23	13	.	46	56	2E	I	73	111	49
Ctrl T	20	24	14	/	47	57	2F	J	74	112	4A
Ctrl U	21	25	15	0	48	60	30	K	75	113	4B
Ctrl V	22	26	16	1	49	61	31	L	76	114	4C
Ctrl W	23	27	17	2	50	62	32	M	77	115	4D
Ctrl X	24	30	18	3	51	63	33	N	78	116	4E
Ctrl Y	25	31	19	4	52	64	34	O	79	117	4F
Ctrl Z	26	32	1A	5	53	65	35	P	80	120	50

(continuación)

Tecla(s)	Dec	Oct	Hex	Tecla(s)	Dec	Oct	Hex	Tecla(s)	Dec	Oct	Hex
Q	81	121	51	a	97	141	61	q	113	161	71
R	82	122	52	b	98	142	62	r	114	162	72
S	83	123	53	c	99	143	63	s	115	163	73
T	84	124	54	d	100	144	64	t	116	164	74
U	85	125	55	e	101	145	65	u	117	165	75
V	86	126	56	f	102	146	66	v	118	166	76
W	87	127	57	g	103	147	67	w	119	167	77
X	88	130	58	h	104	150	68	x	120	170	78
Y	89	131	59	i	105	151	69	y	121	171	79
Z	90	132	5A	j	106	152	6A	z	122	172	7A
[	91	133	5B	k	107	153	6B	{	123	173	7B
\	92	134	5C	l	108	154	6C	|	124	174	7C
]	93	135	5D	m	109	155	6D	}	125	175	7D
^	94	136	5E	n	110	156	6E	~	126	176	7E
–	95	137	5F	o	111	157	6F	del	127	177	7F
'	96	140	60	p	112	160	70				

APÉNDICE C

Almacenamiento para un número en punto flotante

El código binario en complemento a dos usado para almacenar valores en número entero se presentó en la sección 1.6. En este apéndice se presenta el formato de almacenamiento binario usado de manera típica en C++ para almacenar números de precisión sencilla y de precisión doble, los cuales se almacenan como puntos flotantes y dobles, respectivamente. De manera colectiva, los valores de precisión sencilla y doble se conocen por lo común como valores en punto flotante.

Como sus contrapartes los números decimales que usan un punto decimal para separar las partes entera y fraccionaria de un número, los números en punto flotante se representan en un formato binario convencional con un punto binario. Por ejemplo, considérese el número binario 1011.11. Los dígitos a la izquierda del punto binario (1011) representa la parte entera del número y los dígitos a la derecha del punto binario (11) representan la parte fraccionaria.

Para almacenar un número binario en punto flotante se usa un código similar a la notación científica decimal. Para obtener este código el formato de número binario convencional se separa en una mantisa y un exponente. Los siguientes ejemplos ilustran números en punto flotante expresados en esta notación científica.

Notación binaria convencional	Notación científica binaria
1010.0	1.01 exp 011
–10001.0	–1.0001 exp 100
0.001101	1.101 exp –011
–0.000101	–1.01 exp –100

En notación científica binaria, el término exp representa exponente. El número binario que se encuentra antes del término exp es la mantisa y el número binario que sigue al término exp es el valor del exponente. Con excepción del número cero, la mantisa siempre tiene un solo 1 a la izquierda seguido inmediatamente por un punto binario. El exponente representa una

potencia de 2 e indica el número de lugares que deberá moverse el punto binario en la mantisa para obtener la notación binaria convencional. Si el exponente es positivo, el punto binario se mueve a la derecha. Si el exponente es negativo, el punto binario se mueve a la izquierda. Por ejemplo, el exponente 011 en el número

```
1.01 exp 011
```

Significa mover el punto binario tres lugares a la derecha, así que el número se convierte 1010. El exponente −011 en el número

```
1.101 exp -011
```

significa mover el punto binario tres lugares a la izquierda, así que el número se convierte en

```
.001101
```

Al almacenar números en punto flotante, el signo, la mantisa y el exponente se almacenan de manera individual dentro de campos separados. El número de bits usado para cada campo determina la precisión del número. Los formatos de datos de precisión sencilla (32 bits), precisión doble (64 bits) y precisión extendida (80 bits) son definidos por la Norma 754-1985 del Institute of Electric and Electronics Engineers (IEEE; Instituto de Ingenieros Eléctricos y Electrónicos) para que tengan las características dadas en la tabla C.1. El formato para un número en punto flotante de precisión sencilla se ilustra en la figura C.1.

Tabla C.1 Especificación de punto flotante en la Norma 754-1985 del IEEE

Formato de datos	Bits de signo	Bits de mantista	Bits de exponente
Precisión sencilla	1	23	8
Precisión doble	1	52	11
Precisión extendida	1	64	15

El bit de signo mostrado en la figura C.1 se refiere al signo de la mantisa. Un bit de signo de 1 representa un número negativo y un bit de firma 0 representa un valor positivo. En vista que todas las mantisas, excepto por el número 0, tienen un 1 a la izquierda seguido por sus puntos binarios, estos dos elementos nunca se almacenan de manera explícita. El punto binario reside de manera implícita inmediatamente a la izquierda del bit 22 de la mantisa, y siempre se supone un 1 a la izquierda. El número binario 0 se especifica estableciendo todos los bits de mantisa y de exponente en 0. Sólo para este caso, El bit de mantisa a la izquierda implicado también es 0.

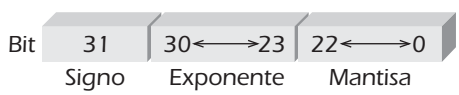


Figura C.1 Formato para el almacenamiento de un número en punto flotante de precisión sencilla.

El campo del exponente contiene un exponente que está sesgado por 127. Por ejemplo, un exponente de 5 se almacenaría usando el equivalente binario del número 132 (127 + 5). Usando ocho bits de exponente, esto se codificaría como 100000100. La adición de 127 a cada ex-

ponente permite que se codifiquen exponentes negativos dentro del campo de exponente sin necesidad de un bit de signo explícito. Por ejemplo, el exponente –011, el cual corresponde a –3, se almacenaría usando el equivalente binario de +124 (127 – 3).

Figura C.2 *La codificación y almacenamiento del número decimal 59.75.*

La figura C.2 ilustra la codificación y almacenamiento del número decimal 59.75 como un número binario de precisión sencilla de 64 bits. El signo, el exponente y la mantisa se determinan como sigue. El equivalente binario convencional de

```
-59.75
```

es

```
-111011.11
```

Expresado en notación científica binaria esto se convierte en

```
-1.1101111 exp 101
```

El signo de menos se representa estableciendo el bit de signo en 1. El 1 a la izquierda de la mantisa y el punto binario se omiten y el campo de mantisa de 23 bits se codifica como

```
11011110000000000000000
```

La codificación del campo de exponente se obtiene añadiendo el valor del exponente de 101 a 1111111, el cual es el equivalente binario del valor sesgado 127_{10}:

```
  1 1 1 1 1 1 1   = 127(10)
        + 1 0 1   =   5(10)
-----------------------------
1 0 0 0 0 1 0 0   = 132(10)
```

APÉNDICE D
Argumentos de línea de comandos

Pueden transmitirse argumentos a cualquier función en un programa, incluyendo la función `main()`. En esta sección se describen los procedimientos para transmitir argumentos a `main()` cuando un programa se invoca inicialmente y hacer que `main()` reciba y almacene en forma correcta los argumentos que se le transmiten. Deben considerarse tanto el lado emisor como el lado receptor de la transacción. Por suerte, la interfaz para transmitir argumentos a una función `main()` se ha estandarizado en C++, así que la emisión y recepción de argumentos puede hacerse de manera casi mecánica.

Todos los programas que se han ejecutado hasta ahora han sido invocados mecanografiando el nombre de la versión ejecutable del programa después que se despliega el indicador del sistema operativo. La línea de comandos para estos programas consiste en una sola palabra, la cual es el nombre del programa. Para computadoras que usan el sistema operativo UNIX, el indicador por lo general es el símbolo `$` y el nombre ejecutable del programa es `a.out`. Para estos sistemas, la línea de comandos simple

```
$a.out
```

comienza la ejecución del programa del último programa fuente compilado que reside actualmente en `a.out`.

Si está usando un compilador C++ en una PC compatible con IBM, el indicador del sistema operativo equivalente de manera típica es `C:\>`, y el nombre del programa ejecutable de manera típica es el mismo nombre que el del programa fuente con una extensión `.exe` en lugar de una extensión `.cpp`. Suponiendo que se está usando una PC compatible con IBM con el indicador del sistema operativo `C:\>` la línea de comandos completa para ejecutar un programa ejecutable llamado `pgmD-1.exe` es `C:\> pgmD-1`. Como se ilustra en la figura D.1, esta línea de comandos causa que el programa `pgmD-1` comience a ejecutarse con su función `main()` pero no se transmiten argumentos adicionales a `main()`.

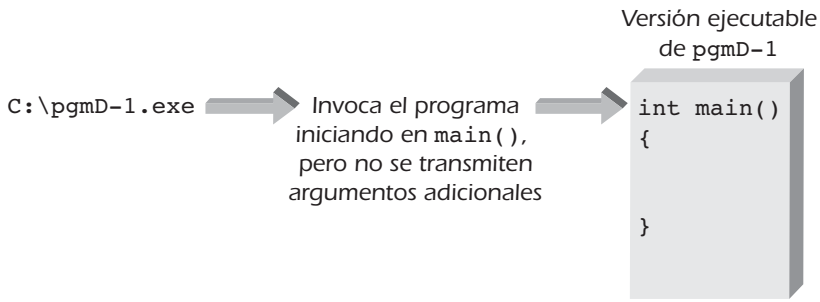


Figura D.1 *Invocación del programa* `pgmD-1.exe`.

Ahora suponga que se desea transmitir los tres argumentos de cadena separados `tres ratones ciegos` en forma directa a la función main de `pgmD-1`. Enviar argumentos a una función `main()` es muy fácil. Se logra al incluir los argumentos en la línea de comandos usada para comenzar la ejecución del programa. Debido a que los argumentos se mecanografían en la línea de comandos se llaman, naturalmente, argumentos de línea de comandos. Para transmitir los argumentos `tres ratones ciegos` en forma directa a la función `main()` del programa `pgmD-1`, sólo se necesita agregar las palabras deseadas después del nombre del programa en la línea de comandos:

```
C:\>pgmD-1 tres ratones ciegos
```

Al encontrar esta línea de comandos, el sistema operativo almacena las cadenas después del indicador como una secuencia de cuatro cadenas (algunos sistemas también almacenan el indicador como parte de la primera cadena). La figura D.2 ilustra el almacenamiento de éstos, suponiendo que cada carácter usa un byte de almacenamiento. Como se muestra en la figura, cada cadena termina con el carácter `null` estándar de C++ `\0`.

Figura D.2 *Los argumentos de línea de comandos almacenados en la memoria.*

Enviar argumentos de línea de comandos a `main()` siempre es así de simple. Los argumentos se mecanografían en la línea de comandos y el sistema operativo los almacena en forma correcta como una secuencia de cadenas separadas. Ahora se debe manejar el lado receptor de la transacción y hacerle saber a `main()` que se le están transmitiendo argumentos.

Los argumentos transmitidos a `main()`, como todos los argumentos de función, deben declararse como parte de la definición de la función. Para estandarizar la transmisión de argumentos a una función `main()`, sólo se permiten dos elementos: un número y un arreglo. El número es una variable en entero, la cual se nombra de manera convencional `argc` (abreviatura para contador de argumento), y el arreglo es una lista unidimensional, la cual se llama de manera convencional `argv` (abreviatura para valores de argumentos). La figura D.3 ilustra estos dos argumentos. El número entero transmitido a `main()` es el número total de elementos en la línea de comandos. En el ejemplo, el valor de `argc` transmitido a `main()` es 4, el cual incluye el nombre del programa más los tres argumentos de línea de comandos. La lista unidimensional transmitida a `main()` es una lista de apuntadores que contienen la dirección de almacenamiento inicial de cada cadena mecanografiada en la línea de comandos, como se ilustra en la figura D.4.

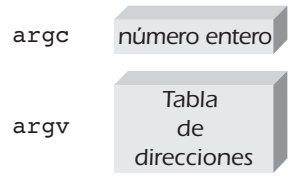


Figura D.3 *Se transmiten un número entero y un arreglo a* `main()`.

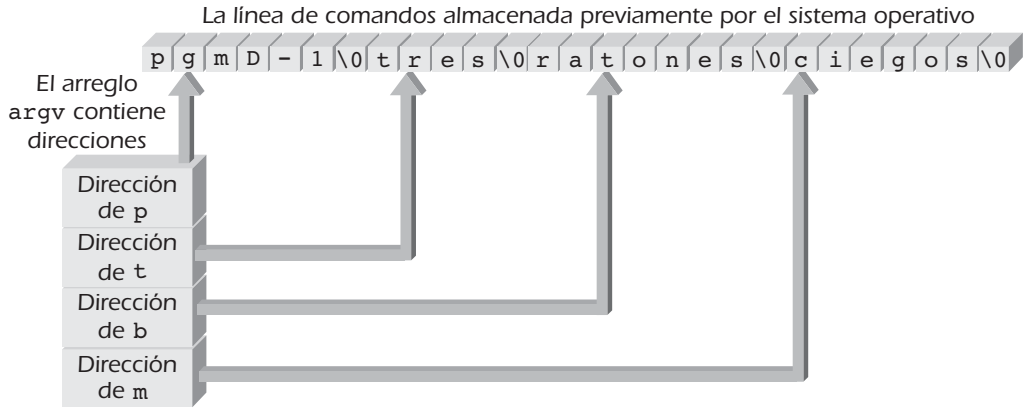


Figura D.4 *Las direcciones se almacenan en el arreglo* `argv`.

Ahora se puede escribir la definición de la función completa para que `main()` reciba argumentos al declarar sus nombres y tipos de datos. Para dos argumentos de `main` los nombres usados de manera convencional son `argv` y `argc`, respectivamente.[1] Debido a que `argc` almacenará un valor en número entero, su declaración será `int argc`. Debido a que `argv` es el nombre de un arreglo cuyos elementos son direcciones que apuntan a donde están almacenados los argumentos de línea de comandos reales, su declaración apropiada es `char *argv[]`. Esto no es más que la declaración de un arreglo de apuntadores. Se lee "`argv` es un arreglo cuyos elementos son apuntadores a caracteres". En resumen, el encabezado de la función completo para una función `main()` que recibirá argumentos de línea de comandos es:

```
int main(int argc, char *argv[])
```

Sin importar cuántos argumentos se mecanografíen en la línea de comandos, `main()` sólo necesita las dos piezas de información estándares proporcionadas por `argc` y `argv`; el número de elementos en la línea de comandos y la lista de direcciones iniciales que indican dónde se almacena en la actualidad cada argumento.

El programa D.1 verifica la descripción al imprimir los datos transmitidos en realidad a `main()`. La variable `argv[i]` usada en el programa D.1 contiene una dirección. Ésta es la dirección que es desplegada por la primera instrucción `cout` dentro del ciclo `for`. Para facilitar la lectura de la salida, esta dirección es moldeada en un valor entero. La notación de cadena `*argv[i]` en el segundo flujo `cout` se refiere a "el carácter al que apunta" la dirección en `argv[i]`.

[1]Estos nombres no se requieren, y puede usarse en su lugar cualquier identificador válido en C++.

Programa D.1

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
  int i;

  cout << "\nEl numero de elementos en la linea de comandos es "
       << argc << endl << endl;
  for(i = 0; i < argc; i++)
  {
    cout << "La direccion almacenada en argv[" << i <<"] es "
         << int(argv[i]) << endl;   // despliega la dirección como un número entero
    cout << "El carácter al que apunta es " << *argv[i] << endl;
  }

  return 0;
}
```

Suponiendo que la versión ejecutable del programa D.1 se llama `pgmD-1.exe`, una muestra de la salida para la línea de comandos `C:\>pgmD-1` es:

```
El numero de elementos en la linea de comandos es 4

La direccion almacenada en argv[0] es 3280388
El carácter al que apunta es p
La direccion almacenada en argv[1] es 3280395
El carácter al que apunta es t
La direccion almacenada en argv[2] es 3280401
El carácter al que apunta es b
La direccion almacenada en argv[3] es 3280407
El carácter al que apunta es m
```

Es evidente que las direcciones desplegadas por el programa D.1 dependen de la máquina utilizada para ejecutar el programa. La figura D.5 ilustra el almacenamiento de la línea de comandos desplegada por la muestra de salida. Como se anticipaba, las direcciones en el arreglo `argv` "apuntan" a los caracteres iniciales de cada cadena mecanografiada en la línea de comandos.

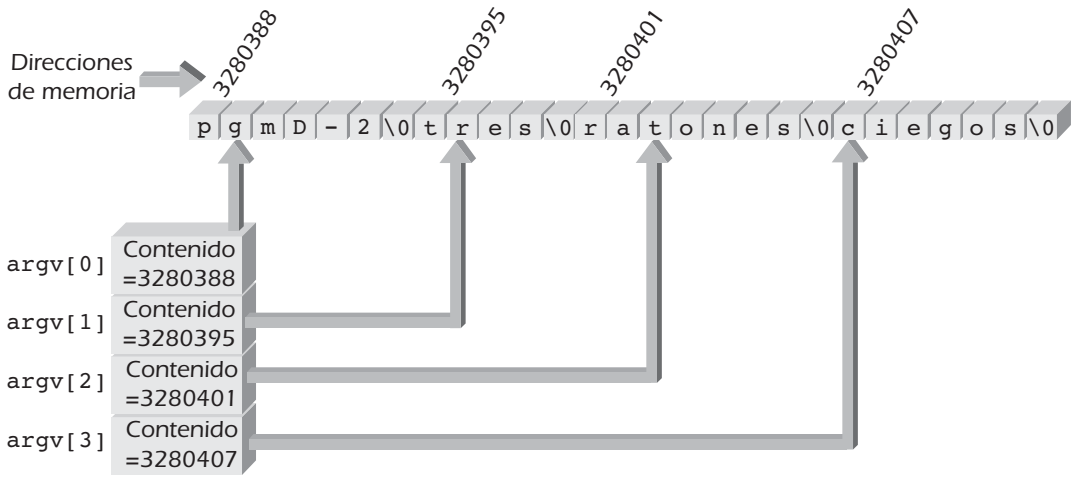
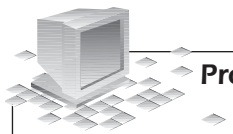


Figura D.5 *La línea de comandos almacenada en la memoria.*

Una vez que son transmitidos los argumentos de línea de comandos a un programa en C++, pueden usarse como cualquier otra cadena en C. El programa D.2 causa que sus argumentos de línea de comandos sean desplegados desde adentro de `main()`.



Programa D.2

```
// Un programa que despliega sus argumentos de línea de comandos
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
  int i;

  cout << "\nLos siguientes argumentos fueron transmitidos a main(): ";
  for (i = 0; i < argc; i++)
    cout << argv[i] << " ";
  cout << endl;

  return 0;
}
```

Suponiendo que el nombre de la versión ejecutable del programa D.2 es `pgmD-2.exe`, la salida de este programa para la línea de comandos `C:\>pgmD-2 tres ratones ciegos` es:

```
Los siguientes argumentos fueron transmitidos a main(): pgmD-2
tres ratones ciegos.
```

Se debe notar que cuando se insertan las direcciones en `argv[]` en el flujo cout en el programa D.2, se despliegan las cadenas en C que apuntan a estas direcciones. Como se mencionó antes, esto ocurre debido a que `cout` desreferencia en forma automática estas direcciones y ejecuta la indirección requerida para localizar la cadena real que se despliega.

Se amerita un comentario final sobre los argumentos de línea de comandos. Cualquier argumento mecanografiado en una línea de comandos se considera una cadena en C. Si se desea transmitir datos numéricos a `main()`, depende de usted convertir la cadena transmitida en su contraparte numérica. Sin embargo, esto rara vez es un problema, en vista que la mayor parte de los argumentos de línea de comandos se usan como banderas para transmitir señales de control de procesamiento apropiados para un programa invocado.

ÍNDICE

◆ A
abstractos, tipo de datos (TDA), 496, 498, 499
acceso a archivos, 480
acceso aleatorio a archivos, 480
acceso secuencial, 480
acceso, especificaciones de, 587
acceso, función de, 520
acumulación, 113
acumulación, instrucción de, 113
agregados, tipos de datos, 612
alcance, 361
  bloque, 198
  clase, 547
  global, 361
  local, 361
  variable, 199, 361
alcance de bloque, 198
alcance de clase, 548
alcance global. 361
alcance local, 361
alcance, operador de resolución de, 364
aleatorios, números, 350
algoritmo para resolver problemas, 342
algoritmos, 17
  de búsqueda, 653
  de ordenamiento, 660
American Standard Code for Information Interchange (ASCII), 52
amiga, función, 553, 564
amigos, lista de, 553
análisis básico, 85
análisis extendido, 85
AND (&&), operador, 189, 792
apertura de un archivo, 447, 473, 475
aplicación, software de, 5
apuntador a estructura (->), operador, 727
apuntador, 669
  aritmético, 689
  como miembros de estructura, 735
  como nombre de arreglo, 680
  direcciones y, 668
  disminuir un, 690
  errores, 707
  incrementar, 690
  inicialización, 692
  notación avanzada, 702
  referencia y, 673
  transmisión, 726
  transmisión de direcciones, 693
apuntador, declaración de, 67l
apuntadoras, variables, 669
archivo binario, 446
archivo estándar de entrada, 468
archivo estándar de salida, 468
archivo, 444
  errores, 488
archivo, apertura de un, 447, 473
  múltiple, 475
archivos de encabezado, 39
archivos múltiples, apertura, 475
argumento(s), 38, 302
  arreglo como, 635
  por omisión, 311
  flujo de archivo como función, 484
  de funciones, 302
  estructuras como función, 723
aritmética con apuntadores, 689
arranque (boot), 6
arreglo(s), 605
  asignación dinámica, 686
  bidimensionales, 620
  búsqueda y ordenamiento, 653
  clase de vector STL, 644
  como argumentos, 635
  componentes de, 608
  dimensionales mayores, 624
  elementos de, 608
  errores, 651
  estructuras, 718
  inicialización, 617
  transmisión de, 699
  unidimensionales, 606
  unidimensionales, 606
  valor índice, 608
  valor subíndice, 608
arreglo, nombre de, 605
  como apuntador, 680
arreglos dimensionales mayores, 624
arreglos, inicialización de, 617
ASCII (American Code for Information Interchange), 52
asignación de memoria, 77
asignación dinámica de arreglos, 686
  asignación de memoria, 686
  estructuras, 741
asignación miembro por miembro, 538
asignación, 538
  constructores de copia, 542
  inicialización base/miembro, 545
asignación, expresión de, 109
asignación, instrucciones, 72
  instrucciones de acumulación, 113
  instrucciones de conteo, 115
asignación, operaciones de, 106
  acumulación, 113
  coerción, 110
  conteo, 115
  errores, 177
  variaciones de asignación, 111
asignación, operadores de, 109, 113, 538
asignación, variaciones de, 111
asociatividad, 66
atómica, variable, 605
atómico, valor de datos, 53
atómicos, datos, 53, 73
atómicos, tipo de datos, 53
auto(mático), 371
auto, clase de almacenamiento, 374 821
auto, variables, 371, 374

◆ B
bandera o indicador, 127
base, clase, 585, 586
base/mimbro, lista de inicialización de, 545
biblioteca de clase, 580 822
biblioteca de funciones matemáticas, 137
biblioteca de funciones, 137
biblioteca de programa, 580
biblioteca personal, 431
bidimensionales, arreglos, 620
  arreglos dimensionales mayores, 624
binaria, búsqueda, 656
binaria, sustracción, 65
binarios, operadores, 62
bisección, método, 759, 765
  regula falsi modificado, 768
  regula falsi, 766
  resumen de algoritmos basados en, 773
bit, 26
bit, operaciones de, 791
  elementos de datos de diferentes tamaños, 800
  operador AND, 792
  operador de complemento, 799
  operador OR exclusivo, 797
  operador OR inclusivo, 795
  operadores de desplazamiento, 801
booleano, tipo de datos, 55, 203
booleanos, datos, 55, 203
bootstrap loader, 6
break, instrucción, 259
bug, 14, 182
búsqueda lineal, 653
búsqueda secuencial, 653
búsqueda, algoritmos de, 653
  binarios, 656
  lineales, 653
bytes, 26

◆ C
C++
  desarrollo de, 6
  errores, 100
  estilo de programación, 45
  estilo de inicialización, 516
  introducción, 34
  operaciones aritméticas, 62
  tipos de datos, 50
  variables e instrucciones de declaración, 70
cabos, 310
cadena C, como nombre de archivo, 452 821

cadena, variable de, 452, 453 826
caja de valores, 27 826
calificador const, 161 825
campo de datos, 712 822
carácter de escape, 53 822
carácter de nueva línea fantasma, 397, 418
carácter, 52
  escape, 53
  null, 618
carácter, archivos basados en, 446
  archivos estándar en dispositivos, 468
  lectura de un archivo de texto, 462
  lectura y escritura, 459
  otros dispositivos, 468
caracteres, código de, 27
caracteres, manipulación de, 410
  E/S, 414
  nueva línea fantasma, 418
  validación de entradas del usuario, 421
case, 217
centinelas, 258
cerrar un archivo, 447, 454
char, 52, 398, 416 822
ciclo de condición variable, 241
ciclo de cuenta fija, 240
ciclo de prueba posterior, 239
ciclo de prueba preliminar, 239
ciclo for, 263, 271
ciclo infinito, 244
ciclo interior, 284
ciclos
  anidados, 284
  ciclo infinito, 244
  ciclos do while, 289
  controlados a la entrada, 239
  controlados a la salida, 239
  cuenta fija, 240
  de condición variable, 241
  estructuras básicas, 238
  exterior, 284
  for, 263
  interior, 284
  prueba posterior, 239
  prueba preliminar, 239
  técnicas de programación, 276
  while, 241
  while interactivos, 250
ciclos anidadas, 284
ciclos controlados a la entrada, 239
ciclos while interactivos, 250
  centinelas, 258
  instrucciones break y continue, 259
  instrucción null, 261
cin, 147 822
clase de almacenamiento, 370, 377 826
clase de vector de la Biblioteca Estándar de Plantillas. Véase clase de vector STL
clase de vector STL, 644
clase, 34-35, 499, 509
  alcance, 547
  asignación, 538
  biblioteca, 580
  cadena, 392
  características adicionales, 547
  construcción, 499
  constructora, 502, 513, 514
  conversión, 537, 568, 570, 572
  destructora, 520
  errores, 533
  especificado de acceso, 587
  funciones, 537
  herencia, 585
  implementación, 503
  miembros, 499
  miembros estáticos, 548
  tipos de datos abstractos, 495
clase, tipo de datos de, 50
clases de almacenamiento de variables globales, 370, 374
clases de almacenamiento de variables locales, 370
código de números, 27
código fuente, 3
coerción, 110
colocación de instrucciones, 162, 308
comandos preprocesadores, 39
comentarios, 46
comillas dobles, para encerrar cadenas, 39, 41, 392
comillas sencillas, 52, 54, 56
compilador, 3
complemento a dos, números en, 27
complemento, operador de, 799
componente de arreglo, 608
comportamiento, 510
comprobación de archivos, 471
compuestas, instrucciones, 196
condición posterior, 307
condición previa, 307
condiciones, expresiones, 187, 231
conexión exitosa, comprobar, 458
const, 161
constante, 51
  simbólica, 161, 162
constantes nombradas, 162
constantes simbólicas, 161, 162
constructor, 502, 513, 514
  conversión, 567
  conversión del tipo, 567
  de copia, 542
  de copia por omisión, 542
  destructor, 520
  en línea, 516
  función operador, 567
  por omisión, 514
  que llama, 516
  sobrecargado, 516
constructor de conversión, 567
constructor de conversión de tipo, 567, 568
constructor de copia, 542
constructor en línea, 516
constructores que llaman, 516
conteo, instrucción de, 115
continue, instrucción, 259
controlado a la salida, 239
controladores de dispositivo, 491
conversiones de tipo de datos, 567
cout, 38, 122, 129
criterios de selección, 186
  operadores lógicos, 189
  problema de precisión numérica, 191
cuerpo de una función, 38, 304

◆ D

declaración hacia adelante, 573
declaraciones múltiples, 75
declaraciones, 70
  asignación de memoria, 77
  como instrucción de definición, 72
  colocación de, 74
  de apuntadores, 671
  dirección de la variable, desplegar la, 79
  hacia adelante, 573
  función, 300
  múltiples, 75
  parámetros, 300
decremento, operador (--), 117
defensiva, programación, 222
definición de una función, 304
definición, instrucción de, 72, 78
delete, operador, 686, 741
depuración, 88, 182
depurador, 183
derivación protegida, 586
derivada, clase, 585, 586
desarrollo, 85
desplazamiento a la derecha (>>), operador de, 802
desplazamiento a la izquierda (<<), operador de, 801
  operador de, 801
desplazamiento aritmético a la derecha, 804
desplazamiento lógico, 804
desplazamiento, 681
desplazamiento, operadores de, 801
desreferencia automática, 676
desreferencia, 673, 676
destructor, 520
desviación, 627
devolución de estructuras, 729
devolución, instrucción de, 323
devolver valores múltiples, 321, 332
diagonal inversa (\), carácter, 41, 53
diagrama de estructura de primer nivel, 12
diagrama de flujo, 20
direccionamiento indirecto, 670
direcciones
  almacenamiento, 669
  apuntadores y, 668
  decremento, 690
  desplegar, 79
  incremento, 690
  transmisión, 693
  uso de, 670
  variables, 79, 669
discriminante, 225
diseño, 85
dispositivos, archivos estándar en, 468
división (/), operador de, 62, 65
división de módulo, 62
do while, ciclos, 289
  verificaciones de validez, 291
do, instrucción, 290
double, tipo de datos, 58

◆ E

ecuación cuadrática, 224
ecuaciones lineales, 224
ecuaciones trascendentales, 757
elemento de arreglo, 608
elemento de entrada, 86

encabezado de función, 38, 304
encapsulamiento, 531, 586
endl, manipulador, 66
enfoque descendente, 85
ensambladores, 3
entero corto, número, 73
entero, número
  división, 65
  expresión, 64
  números, 52
  tipo de datos, 51
  valores, 52, 416
entrada y salida (E/S), manipulación, 414
entrada y salida de valores de arreglo, 611
entrada y salida, rutinas de clase, 395
entradas del usuario, validación de, 152, 153, 421
error de sintaxis, 180
error tipográfico, 180
errores de análisis gramatical, 180
errores en tiempo de compilación, 180
errores en tiempo de ejecución, 180
errores lógico, 180
errores, 22
  archivo, 488
  arreglos, 651
  asignación, 598
  asignación, formateo, entrada interactiva, 177
  C++, programación inicial en, 100
  construcción de clases, 533
  definición y procesamiento de cadenas, 437
  direcciones y apuntadores, 707
  en tiempo de compilación, 180
  encontrar raíces por bisección modificada, 788
  errores de análisis gramatical, 180
  estructuras y uniones, 751
  instrucción de selección, 230
  instrucciones de repetición, 293
  en tiempo de ejecución, 180
  función, 379
  lógicos, 180
  programación, 23
  sintaxis, 180
  tipográficos, 181
escalamiento, 353
escalares, variables, 605
escritura de archivos, 459
espacio de nombres, 431
espacio en blanco, 45
estado, 509
estilo C de inicialización, 516
estilo de programación, 45
structuras, 712
  arreglo de, 718
  asignación dinámica de estructura de datos, 741
  como argumentos de función, 723
  contenido de, 712
  declaración de, 712
  datos, 718
  devolución de, 729
  errores, 251
  instrucción if-else, 193
  instrucciones if anidadas, 206
  instrucción switch, 216
  listas vinculadas, 733
  poblar, 712
  prueba de programación, 234
  selección, 185
  sencillas, 712
  transmisión, 723
  uniones, 748
estructuras de datos heterogéneas, 718
estructuras de datos homogéneas, 718
estructuras de selección, 185
eexcepción, 386
exponencial, función, 170
exponencial, notación, 59
expresión aritmética binaria simple, 62
expresión, 64, 107
relacional, 186
expresiones en modo mixto, 65
exterior, ciclo, 284
extern, 370
extern, clases de almacenamiento, 375
extracción (>>), operador de, 150

◆ F

firmados, tipos de datos, 57
float, 73
flujo de archivos de entrada, 446
  archivos, 444
  cerrar un archivo, 454
  métodos, 447
  objetos, 446
flujo de archivos de salida, 444, 446
  archivo de texto, formatear, 461
flujo de archivos, 446
  como argumentos de función, 484
  mecanismo de transferencia, 490
  métodos, 447
  objetos, 446
flujo de control, 185
flujo secuencial, 185
flujo, 446
for, instrucción, 263
formateo
  errores, 177
  números para salida del programa, 122
formateo en memoria, 493
fórmula, 20
FORTRAN (FORmula TRANslation), 496
fstream, 446, 456
  métodos, 489
  objetos, 456
fuga de memoria, 688
función controladora, 37
función en línea, 327
función llamada, 301
función miembro, 499, 500
función operador de conversión, 567, 570, 572
función que llama, 301
función, 34, 300
  amiga, 553
  argumentos de, estructuras como, 723
  cabos, 309
  con listas de parámetros vacías, 311
  cuerpo de la, 38, 304
  de acceso, 520
  definición, 304
  errores, 379
  encabezado, 38, 304
  nombres, 35, 312
  operador, 558
  operador de conversión, 570, 572
  plantillas, 313
  prototipo, 302, 306
  reutilización de nombres, 312
  sobrecarga, 312
  vínculo de la, 595
funciones de biblioteca estándares, 580

◆ G

get( ), método, 467
getline( ), 466

◆ H

hardware, 24
herencia de clase, 585
herencia múltiple, 585
herencia simple, 585
herencia, 585, 586
hexadecimales, números, 131
hija, clase, 585, 586

◆ I

identificadores, 35, 547
if-else, instrucción, 193
ifstream, 446, 456
igualdad (==), operador, 294
implementación, 509
impresión en eco, 183
include, instrucción, 39
inconsistencia notacional, 417
incremento (++), operador de, 116
incrustados, nombres de archivos, 452
indexada, variable, 608
indicador de comandos, 149
índice, 608
indirección (*), operador de, 670
ingeniería de software, 10
inicialización en tiempo de ejecución, 372
inicializada, 76
  apuntador, 692
  arreglo, 617
  base/miembro, 545
  en tiempo de ejecución, 372
  estilo C, 516
inserción (<<), operador de, 63
instanciación, 509
instancias, 509
instrucciones
  #include, 39
  break, 259
  colocación, 162, 308
  compuestas, 196
  continue, 259
  de asignación, 72, 113, 115
  de declaración, 72
  de definición, 72, 78
  de repetición, 237
  de devolución, 323
  do, 290
  for, 263
  if-else, 193
  if anidadas, 206
  null, 261

switch, 216
while, 241
instrucciones if anidadas, 206
int, 52
integración numérica, 778
integración, 778
integrado a
conversión integrada, 567
conversión de clase, 568
integrado, tipo de datos, 50, 498
interactivos, ciclos while, 250
intercambio (burbuja), ordenamiento por, 664
intercambio, ordenamiento por, 664
interfaz, 509
intérprete, 3
invocación, 4
iostream, 446
iostream, biblioteca de clase, 490
componentes, 491
formateo en memoria, 493
mecanismo de transferencia de flujo de archivos, 490
iteración, 14, 238

◆ J
jerarquías de clase, 586
jerarquías, 586
justificación a la derecha, 128

◆ L
lanzar una excepción, 386
lectura de archivos, 462
lenguaje compilado, 3
lenguaje ensamblador, 2, 3
lenguaje híbrido, 496
lenguaje interpretado, 3
lenguaje orientado a procedimientos, 4, 496
lenguajes
de nivel alto, 3
de nivel bajo, 3
de máquina, 2
ensambladores, 2, 3
orientados a objetos, 4
orientados a procedimientos, 4
lenguajes basados en objetos, 586
lenguajes de nivel alto, 3
lenguajes de programación, 2, 4
lenguajes orientados a objetos, 4, 496, 586
línea nueva, secuencia de escape, 54
listas de parámetros, vacía, 311
listas vinculadas, 733
literal, 51
llamada por referencia, 332
llamada por valor, 321
llaves, colocación de, en una instrucción compuesta, 199
ciclos for, 269
long double, tipo de datos, 58
lvalue, 111

◆ M
main( ), función, 37
manejo de excepciones, 386, 471
manipulador parametrizado, 130
manipulador, 64
de flujo, 122
setiosflags, 128
setprecision, 126
setw, 124
máquina, lenguaje de, 2
máscara, 793
mayor que o igual a, operador (>=), 187
mayor que, operador (>), 187
memoria, asignación dinámica de, 686
menor que (<), operador, 187
método científico, 10
métodos numéricos, 755
integración numérica, 778
errores, 788
método de bisección, 759
método de la secante, 755
refinamientos del método de bisección, 765
obtener raíces, 756
regla de Simpson, 784
regla trapezoidal, 779
métodos, 444
microprocesador, 25
miembros de datos, 499
mnemónicos, 36
modo de lectura, 449
modo, 447
modularidad con el uso de funciones, 299
alcance variable, 361
clases de almacenamiento de variables, 370
declaraciones de funciones y parámetros, 300
devolver un solo valor, 321
devolver valores múltiples, 332
errores, 379
módulo (%), operador de, 65
módulos, 34
molde, operador, 142
moldes, 142
multiplicación (*), operador de, 62
multiprogramados, 6
multitareas, 6
multiusuario, 6
mutador, método, 521

◆ N
negación (-), operador de, 65
negación, 65
new, operador, 686, 741
nivel bajo, lenguajes de, 3
no firmado, tipo de datos, 57
nombre externo de archivo, 444
nombres de archivo incrustados e interactivos, 452
identificación de, 465
nombres de archivo interactivos, 452
NOT (!), operador, 190
notación de la O grande, 660
nula, instrucción, 261
null, apuntador, 734
nulo, carácter, 618
números mágicos, 161
números, formato para la salida del programa, 122

◆ O
objeto de archivo físico, 468
objeto de archivo lógico, 468
objeto(s), 507, 509
obtención de raíces, 756
octales, números, 131
ocultamiento de datos, 502
ocultamiento de información, 509
ofstream, 446, 456
omisión, argumento por, 311
omisión, constructor de copia por, 542
omisión, constructor por, 514
opciones de carrera, consideración de las, ingeniería aeronáutica y aerospacial, 29
ingeniería química, 382
ingeniería civil, 235
ingeniería eléctrica, 103
ingeniería industrial, 297
ciencia de materiales e ingeniería metalúrgica, 438
ingeniería mecánica. 183
opcode, 2
operaciones aritméticas, 62
asociatividad, 66
tipos de expresión, 64
división de enteros, 65
negación, 65
precedencia, 66
operador de adición (+), 62
operador, funciones, 558, 559
como amigas, 564
conversión, 567
operadores
adición, 62
AND, 189, 792
apuntador a una estructura, 727
aritméticos, 62
asociatividad, 66
bit, 791
complemento, 799
conversión, 567, 570, 572
de asignación, 109, 113, 538
de decremento, 117
de residuo, 65
delete, 686, 741
desplazamiento, 801
desplazamiento a la derecha, 802
desplazamiento a la izquierda, 801
división, 62, 65
extracción, 150
igualdad (==), 294
incremento (++), 116
indirección, 670
inserción (<<), 63
lógicos, 189
mayor que (>), 187
mayor que o igual a (>=), 187
módulo, 65
molde, 142
multiplicación, 62
menor que (<), 187
negación, 65
new, 686, 741
NOT (!), 190
OR, 189, 795, 797
precedencia de los, 67
prefijo para decremento, (--), 117
prefijo para incremento, (++), 116
relacionales, 186
resolución, 364
resolución de alcance, 364

sizeof( ), 55
sobrecargados, 540
sufijo para decremento (--), 117
sufijo para incremento (++), 116
sustracción, 62
unario, 65
operadores aritméticos, 62
operadores lógicos, 189
AND (&&), 189
NOT (!), 190
OR (||), 189
operando, 62
OR (||), operador, 189
exclusivo, 797
inclusivo, 795
OR exclusivo, operador (^), 797
OR inclusivo (|), operador, 795
ordenamiento externo, 660
ordenamiento interno, 660
ordenamiento(s), 660
burbuja, 664
externo, 660
intercambio, 664
interno, 660
selección, 660
ordenamiento, algoritmos de, 660
organización de archivos, 480
organización secuencial, 480
orientaciones
a objeto, 4
a procedimiento, 4

◆ P

padre, clase, 585, 586
palabra clave, 35
palabras, 28
parámetro, 300
parámetros formales, 304
pila, 327
poblar la estructura, 712
polimorfismo, 586, 593
precedencia, 67
precisión doble, número de, 58
precisión numérica, 191
precisión sencilla, número de, 58
precisión, 60
prefijo de plantilla, 314
prefijo para decremento (--), operador, 117
prefijo para incremento (++), operador, 116
privados, derechos de acceso, 502, 587
procedimiento de desarrollo de software, 10, 85
procedimientos, 4
programa de computadora, 2
programa ejecutable, 2
programa fuente, 3
programa objeto, 4
programación, 2
algoritmos, 17
errores, 22
hardware de computación y conceptos de almacenamiento, 24
introducción a la, 2
orientada a objetos, 4
orientada a procedimientos, 4
solución de problemas y desarrollo de software, 9
programas modulares, 34
programas robustos, 152
protegidos, derechos de acceso, 587
prototipo de función, 302, 306
prueba de aislamiento, 310
prueba de la programación, 234
pruebas, 88, 234
públicos, derechos de acceso, 502, 587
punto de información
apertura de archivo, 473
arreglos y vectores, 647
bibliotecas de programa y de clase, 580
bloques try, 478
cadenas C como nombres de archivo, 452
ciclos for y while, 271
clases de almacenamiento, 377
condiciones previas y condiciones posteriores, 307
conexión exitosa, comprobar una, 458
constructores, 514
datos atómicos, 53, 73
datos del flujo de salida de un archivo de texto, 461
datos en el flujo cout, 129
definiciones y prototipos de función, 306
encapsulamiento, 531
estructuras de datos homogéneas y heterogéneas, 718
flujos de entrada y salida, 446
funciones de acceso, 520
indicador o bandera, 127
interfaces, implementaciones y ocultamiento de información, 509
lenguajes basados en objetos y orientados a objetos, 586
lenguajes de procedimientos, híbridos y puros orientados a objetos, 496
lvalue y rvalue, 111
llaves, colocación en una instrucción compuesta, 199
llaves, colocación en ciclos for, 269
manipulador endl, 66
método mutador, 521
métodos get( ) y putback( ), 467
método put( ), 462
'\n' y "\n", 56
nombre y ubicación de un archivo, 465
objetos fstream, 456
precisión, 60
programas orientados a objetos y orientados a procedimientos, 57
prueba de aislamiento, 310
sintaxis, 40
tipos de datos string y char, 398
tipos de datos agregados, 612
tipo de datos booleano, 203
tipo de datos char y valores en números enteros, 416
valores e identidades, 547
punto flotante, expresión, 64
punto flotante, número, 58
put( ), método, 462
putback( ), método, 467

◆ R

rastreo del programa, 182
real, expresión, 65
real, número, 58
referencias desreferenciadas implícitamente, 673
referencias, 333
apuntadores y, 673
variables, 674
register, clase de almacenamiento, 370, 374
registro, 712
registros, 374
regula falsi, método, 766, 768
relacional simple, expresión, 186
relacionales, expresiones, 186
relacionales, operadores, 186
operadores lógicos, 189
simples, 186
valor de, 187
relajación, factor, 772
repetición, instrucción de, 238
ciclos anidados, 284
ciclos do while, 289
ciclos for, 263
ciclos while, 241
estructuras básicas del ciclo, 238
errores, 293
técnicas de programación con ciclos, 276
requerimiento de programa, 10
residuo (%), operador de, 65
residuo de potencias, método, 357
resolución, operador de, 364
resultados numéricos usando cout, 122
rvalue, 111

◆ S

salida formateada, 122
salida, formatear, 122
secante, método de la, 775
sección de implementación, 503
secuencia de escape, 54
secuencia, 14
secundario, almacenamiento, 24
seekg( ), 481
seekp( ), 481
selección unidireccional, 199
selección, 14
selección, ordenamiento por, 660
sencillas, estructuras, 712
sensible al uso de mayúsculas y minúsculas, 36
setprecision( ), 126
setw( ), 124
seudoaleatorios, números, 351
seudocódigo, 20
signo de número (#), 39
signo de porcentaje (%), 65
signo ponderado, 28
Simpson, regla de, 784
sintaxis, 40
sistema operativo, 6
sistema, software de, 5
sistemas, enfoque de, 10
sizeof( ), operador, 55
sobrecargadas, funciones, 139, 313, 558
sobrecargado, constructor, 516

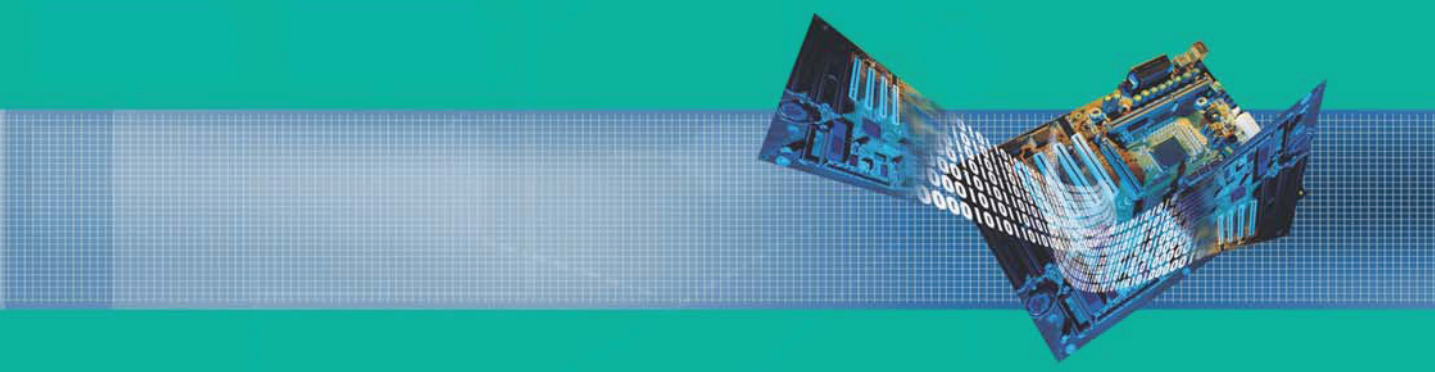
sobrecargados, operadores, 540
software, 2
  de aplicación, 5
  de sistema, 5
sqrt( ), 138
static, clase de almacenamiento, 370
static, miembros de clase tipo, 548
static, variables, 372
string, clase, 392
  carácter de nueva línea fantasma, 397
  entrada y salida, 395
  funciones, 393
  métodos de procesamiento, 399
  tipo de datos, 398
subclase, 585
subindexada, variable, 608
subíndice, 608
sufijo para decremento (--), operador, 117
sufijo para incremento (++), operador, 116
superclase, 585
sustracción (-), operador, 62
switch, instrucción, 216

◆ T
tamaño del almacenamiento, 55
tellg( ), 481
tellp( ), 481
texto, archivos de, 446
  formatear los datos del flujo de salida de, 461
  lectura, 462
tipos de datos, 50, 498
  abstractos, 496
  agregados, 612
  char, 398
  conversiones de, 567
  double, 58
  notación exponencial, 59
  punto flotante, 58
  entero, 51
  long double, 58
  tamaño del almacenamiento, 55
  cadena, 398
tipos primitivos, 50
token, 49
transmisión a través del ciclo, 238
transmisión de apuntador, 726
transmisión de arreglos, 699
transmisión de direcciones, 693
  notación avanzada para apuntadores, 702
  transmisión de arreglos, 699
transmisión de estructuras, 723
transmisión por referencia, 332, 694
transmisión por valor, 321, 332
trapezoidal, regla, 779
truncado, 65
try, bloques, 478

◆ U
ubicación de un archivo, identificación de la, 465
unarios, operadores, 65
unidad central de procesamiento (CPU), 25
unidad de aritmética y lógica (ALU), 24
unidad de control, 24
unidad de entrada y salida, 24
unidad de memoria, 24
unidimensionales, arreglos, 606
  valores de entrada y salida, 611
unión, 748

◆ V
validación
  de entradas, 423
  de entradas del usuario, 152, 153,421
validación de datos de entrada, 222, 423
valor de entrada, 86
valor literal, 51
valor(es), 547
  múltiples, devolver, 332
  un solo, devolver, 321
valores de arreglo, entrada y salida de, 611
variable global, 361
mal uso de, 366
variables globales, clases de almacenamiento, 370, 374
variables locales, 361
variables, 70, 71
  alcance, 199, 361
  atómicas, 605
  clase de almacenamiento, 370
  direcciones de, 79, 669
  escalares, 605
  globales, 361
  indexadas, 608
  locales, 361
  nombrar, 71
  referencia, 674
  subindexadas, 608
vector, 647
verificación de escritorio, 180
verificación de límites, 611
verificaciones de validez, 291
vinculador, 4
vínculo dinámico, 595
vínculo estático, 595
virtuales, funciones, 595
void, 301, 311

◆ W
while, ciclo, 241, 271
while, instrucción, 241



Texto muy necesario para los estudiantes de ingenierías y ciencias que requieran el conocimiento de un lenguaje de programación de alto nivel como C++. Ahora, en la segunda edición, el autor reescribió gran parte del material a fin de presentar una obra más útil y actualizada.

CARACTERÍSTICAS

- El nuevo capítulo *Completando lo básico* (7 en esta edición) analiza las características principales del estándar ANSI/ISO.
- Incluye nuevos estándares de C++.
- Nueva sección sobre formato de números para los resultados de programas.
- Conserva la útil sección *Errores comunes de programación.*

CENGAGE Learning™

http://latinoamerica.cengage.com

ISBN-13: 978-607481437-8
ISBN-10: 607481437-6

9 786074 814378